Low Communication Cost Parallel System Using PCs

By

Yiu Sau Yan Vincent

Supervised by: Dr. Chi Chi Hung &
Dr. Young Ho Fai Gilbert

Submitted to Department of Computer Science and Engineering
in partial fulfillment of the requriement for the

Master of Philosphy

at the Chinese University of Hong Kong

June-1996

To John & Bing ...

Abstract

This project is to design and implement a low cost multiple computer model for parallel and distributed parallel applications. The system consists of a number of Unix machines connected by a network. In the past decades, vast research effort has been done on multiple computer systems. Our system is a *message passing* parallel system. To ensures that the system is cost effective and practical, off-the-shelf hardware have been employed. In the system, we have been concentrated on reducing the *communication cost.* Mechanisms have been employed to reduce the cost in passing local messages (communication between processes on the same host) and network messages (communication between processes on different hosts.) On selected applications, Our platform shows a fivefold performance improvement over its PVM counterparts.

Contents

List of Figures

Chapter 1

Introduction

This project is to design and implement a low cost multiple computer mode for parallel and distributed parallel applications. The system consists of a number of Unix machines connected by a network. In the past decades, vast research effort has been done on multiple computer systems. A number of programming packages/tools have been built for exploring the hidden computing power in CPU farms. Both commercial products and research oriented packages are readily available. However, studies has shown that these packages only improve the performance of loosely-coupled parallel applications. Various approaches have been studied aiming at solving the problem. IRI Multicomputer System is designed and implemented to provide a platform for tighter-coupled parallel applications.

Our system is a *message passing* parallel system. To ensures that the system is cost effective and practical, off-the-shelf hardware has been employed. In the system, we are concentrated on reducing the *communication cost.* Mechanisms are employed to reduce the cost in passing local messages (communication between

processes on the same host) and network messages (communication between processes on different hosts.) On selected applications, Our platform shows a fivefold performance improvement over its PVM counterparts.

In the latest implementation of the the multiple computer System, *Pentium PCs* and *SCSI Bus* are used. The Pentium PCs are used as the processing elements. The SCSI Bus is used as the communication bus. The project is divided into three parts:

Communication Protocol - this part of the project is to specify a set of operations to be supported on the the multiple computer system. The protocol is to provide a ground for the implementation.

Multiple Computer Infrastructure - this part of the project involves the design and implementation of a multiple computer infrastructure basing on the latest off-the-shelf hardware. The architecture of the multiple computer system is a general design, which can be applied to hardwares other than those used in this project. However, the latest implementation of the design is system dependent.

System Supporting Facilities - this part of the project involves the design and development of facilities that supports the multiple computer architecture. However, these facilities are not part of the multiple computer infrastructure. They can be used by other programs, though they are developed for the sake of the system implementation.

Chapter 2

Related Works

The discussion on parallel and distributed processing has lasted for decades. The speed of a single processor do not able to catchup up with the ever increasing demand for faster solutions and large size problems. Faster machines are expected in various application areas including simulation, image processing, information processing and artificial intelligence. Despite the huge investments in sequential machines, the possibility of using multiple processors to improve computational power was recognized and being explored.

Technological advancements in the design of parallel processing environment provide the possibility of performance improvements in applications. In the past, there has been significant research on *tightly-coupled* parallel processing systems, better known as *multiprocessors*. Serial applications could now be executed on multiprocessors. Significant research is also concerned with another kind of parallel processing environment: the use of workstations in a *loosely-coupled* environment, or distributed system. A *distributed system* provides the advantages of parallel

execution while providing a scalable growth of number of CPUs. Multiprocessor system restricts that all CPUs must be physically located near one another.

In order for an application's processes to work in parallel, there must be mechanisms to permit the processes to communicate. There are many ways for parallel processes to communicate. In a tightly-coupled system, since all processors are physically close to one another, a typical scheme used for process communication is *shared memory* [DeC89]. Meanwhile, the processors are usually connected by a high speed backplane bus [Cat95]. In distributed systems however, the location of processors is constrained by the distance of the network. The two most common forms of communication that have been used are *message passing* and *remote procedure call* [GeS93]. Recently, however, there has been a move toward the use of shared memory as a communication medium.

In a loosely-coupled system, since each CPU may be located anywhere on the network, it would not be appropriate to have a shared address space centralized in one particular location. To capture the advantages of a loosely-coupled parallel processing environment and use shared memory for communication, a new approach to communication must be used. This approach to interprocess communication is called *distributed shared memory (DSM)* [Gol93, KhN93].

2.1 Tightly-coupled Parallel Systems

A tightly-coupled parallel processing environment consists of several CPUs, one or more memory units and an interconnection network that physically connects the components. A tightly-coupled system is characterized by a single operating system that controls all the hardware.

In a tightly-coupled system, each CPU in the system has the ability to read and write from memory. The primary purpose of a CPU is to execute instruction codes read from memory and to perform arithmetic, logic, and control operations. A limited amount of data can be stored in the CPU's internal registers. The data stored in registers are typically intermediate results and control information. All data and instructions must be read from the memory unit.

The actual memory implementation and connectivity varies widely with the system implementation. Sometimes there exists one shared memory that all processors have access to, or there is memory associated with each processor. In either case, each processor in the system is directly connected to every memory unit and each processor has read and write access to every unit. This connectivity can be achieved through the use of a bus, cross bar switches, or multistage interconnections.

A time-shared bus is a simple method of giving all processors access to the memory storage units. The bus only allows one process to access memory at any give time. Before a process can use the bus, it must first check if the bus is available, and can

only begin transmission when it is given control of the bus. The bus organization makes it easy to add or take away units simply by connection or disconnection. It is simplistic and cost effective. However, a single bus organization is only sufficient for a moderate number of processors since there can only be one transmission on the bus at a given time. A more innovative and efficient design is the use of different levels of buses.

2.2 Loosely-coupled Parallel Systems

A loosely-coupled parallel processing environment consists of two or more independent computer systems connected by a communication link. Each workstation has its own operating system and local storage. These Systems are being used frequently in both industry and in academia because the user can work through a transparent interface to the system [GeB95, BrD94, CaG89, DeC89, MPI94, Paj91]. Network file systems available today can provide an interface which makes it appear to the user as if he/she is the only user of the system.

In a loosely-coupled system, workstations can function independently or they can work cooperatively by communicating via the network. Because the workstations in a loosely-coupled environment are not limited to be physically near to one another, the concept of shared memory has not been widely used as a communication media. This has led to the extensive use of message passing to implement communication among parallel processes. Since the communication media for parallel

processes in tightly-coupled systems is shared memory, applications must be completely re-written before they can be executed in a loosely-coupled system that uses message passing.

Recently work has been focused on distributed shared memory for loosely-coupled systems. The motivation behind these projects was two-fold. First the use of shared memory in a distributed environment will allow applications to be executed on both tightly- and loosely-coupled systems with little or no changes. It is believed that there are considerable number of applications written using the AT&T System V shared memory facility. By developing a distributed shared memory system using AT&T System V interface. These applications can be re-used in the new parallel environment without being restructured and reorganized. The second motivation for exploring a shared memory paradigm is that a larger group of applications can be used, while still gaining the advantage of workstations. Currently, there are many loosely-coupled systems implemented that have not taken advantage of the parallel computing possibilities. The transparent interface can provide a smooth transition into the use of highly parallel applications.

The paradigm focused in this project is a message-passing multiple computer. In this project, Parallel Virtual Machine (PVM) is used as the base for comparison. PVM is a software package that enables the use of networked heterogeneous Unix computers as a single parallel machine denoted as the virtual machine. Detail information and implementation of the package can be found in

[GeB95, GeS93, WhA94]. The software package consists of a daemon program and a C style function library callable by programs to be run on the virtual machine. The virtual machine environment is setup by pvmd daemon. The daemon must exist in every member machine of the virtual machine. It acts as a message router and controller that provides a contact service among the hosts. The daemon does not perform computational work of the application whereas it is responsible for authentication, process control and fault detection.

The function library is a set of user callable PVM interface routine. The routines are used for message passing, spawning processes, coordinating tasks and reconfiguring the virtual machine. The parallelism of applications must be stated explicitly in the programs and the programs must be linked with the PVM library. Application programs can be developed using C, C++ and Fortran.

The member hosts of the virtual machine are usually interconnected by a network such as the Ethernet and FDDI. The hosts must be able to communicate directly using IP protocols (TCP and UDP) [Rag93, Ste94, WrS95].

The PVM applications experience different levels of transparency over the virtual machine. Using the transparent mode, tasks are automatically executed on hosts selected by the pvmd. When it is required to specify a particular architecture for certain tasks, the architecture-dependent mode may be used. Using the low-level mode, the user may also assign tasks to a particular host.

The processes of PVM applications can exercise arbitrary relationships and de-

pendencies among each other. The processes are also allowed to communicate and synchronize with each other at any point of execution. These features of PVM permit a variety of parallel processing modes to be supported. It is obvious that MIMD computational mode is supported. In addition, SPMD and master-slave modes can also be used [DaG88].

In the *master-slave* mode a set of slave processes perform work for one or more master processes. In the *Single Program Multiple Data (SPMD)* all the hosts are running the same program on different data whereas there is no master program controlling the computation. This is close to a SIMD architecture except that the processing elements are not working in lockstep and are not controlled by a control unit.

The architecture of PVM should also support SPMD mode. However, the use of relatively slow and high latency interconnection network as communications subsystem limits the possible granularity of PVM applications. The lower bound of task granularity of PVM applications is limited by the available network bandwidth and the level of fine-grain parallelism required by SIMD usually can not be achieved.

As a result, the range of parallel applications feasible for workstation cluster is limited. The limitations are imposed by the fact that the network and protocol are designed for the transmission of data that are relatively large in size. This problem does exist over most of parallel processing systems that relay on workstation as

processing elements [Sur93].

Recently, research has been on employing different technologies from protocol, hardware to multiple computer architecture to reduce the communication among processors [ChG94, HwM96, Mar94, OkD90]. In our project, we aim at building a *tighter-coupled* multiple computer system starting from the loosely-coupled end.

Chapter 3

Communication Protocol

Message passing based parallel applications usually have some common needs in information exchanging methods and system services. Different standards emphasizing on different expect of distributed and parallel processing is emerging. In this part of the project, we aimed at specifying a set of basic functions and their corresponding interfaces for reference by the developers and users of the system. The functions, in turn, could be used to built other complex functions, depending on the need of the users.

The *CommUnication Protocol (CUP)* specifies a number of functions and their interfaces to be supported in our perform. The objective of the protocol is to

- to reduce confusion in implementation. A number of people uses and take parts in project development. The specification and the standard provides a common ground for the group.
- to speedup application development. Applications could be built based on

the specification, before the system is ready.

- to enhance cross platform compatibility. The programs that uses CUP should not need any changes for porting to platforms that support CUP.

- to provide a base for verification. Test programs could be developed following the specification for verification purpose to detect any incorrectness in the implementation.

3.1 Terminology

message

Programs are communicated using messages. Each message has two part: header and content. The header contains control information. This part is common in every message. The message content depends on how it is interpreted by the programs involved in the communication.

system id

Each process is assigned an system id which is unique over the system. Each system id identifies a process and the system resources used by the process.

communication port

Programs use the communication port to send and receive messages. The messages are transfered in passive form. A client expecting a message has to listen to its port and wait for incoming message.

control transfer/control-level protocol

The communication path between the server programs and user programs for the exchange of commands and replies. This connection follows the ComUnication Protocol.

data transfer/user-level protocol

A full duplex communication over which data is transferred, in a specified mode and type agreed by the user programs. The data transferred is represented in binary format and the content is unknown with respect to the system.

connected

Each process has to attach to the system for using the communication facilities. A process attached to the system is described as CONNECTED.

disconnected

A process after finishing using the communication facilities, has to free the reserved resources for supporting the functions. The status after cleaning up all the resources is described as DISCONNECTED.

cup command

In addition to message passing, the system supports a number of command to be accessed by both local and remote processes. The user processes have to specific the command and provide the parameters required by the command.

connectionless

The communication port is a connectionless port. User programs can use the port to send and receive messages from multiple programs. The sender and the recipient may be the same program.

reliable data transmission

The transmission of data is reliable. That is, after passing the message into the system, the sender may assume that the recipient can get the message error-free.

cup environment The user level environment that supports CUP. The underlying machine architecture is not interested. The hardware may consists of a single processor machine, cluster of workstations or multi-processor supercomputer.

3.2 CUP Model

With the above definitions in mind, the following model (fig 3.1.) diagrammed for the CUP Service.

In the model described in fig. 3.1, the user process initiate the *user-level protocol*. The user process invoke *control-level protocol* to attach to the system via the *control transfer*. The process will than be assigned a *communication port*. The process may now use the port for sending and receiving messages. The process may also invoke a control transfer for *system service*. After finishing all the communications, the program invoke another control transfer for closing all the resource

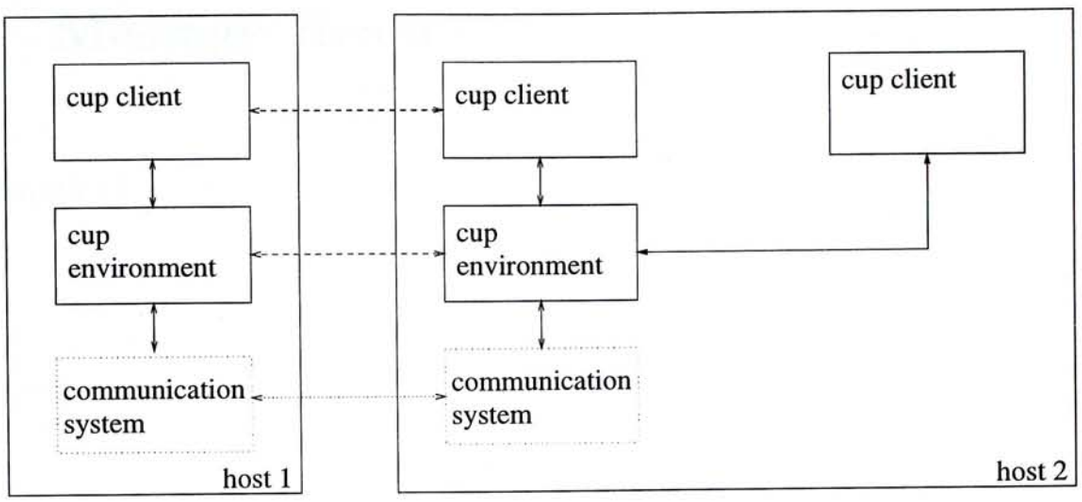


Figure 3.1: Interaction between sender/receiver, device driver and host adapter

and *disconnect* from the system.

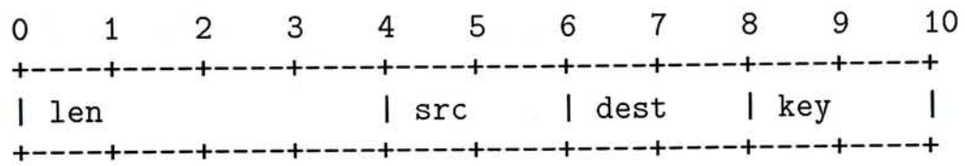
The protocol requires that the client process be attached to the system with the port opened. It is the responsibility of the user to request to detach from the system, while the system will take the action. The system will actively not disconnect a user process.

3.3 Message Format

Each message consists of two parts the *header* and the *content*. The header consists of some control information and message parameters. In control message, the message content of each command is specified. To invoke the commands, the user program must fill the message content following the specification. The message content of user data transfer is in free format.

3.4 Message Header

The length of message header is 10 byte. It is of the follow format:



len - The length of message content. The message size is stored in unsigned format. According to the standard, the maximum size of the message is 0 to $2^{32} - 1$ bytes. The message size could be zero which identifies an empty message. Some of the system message only have the header.

src - senders system id. The system will automatically insert the senders system id into the src field when the process submit the message to the system. Before the an id is assigned, a process has to submit a initial command to the system for getting a valid id. A recipient may also specifies to listen and read messages from a particular sender.

dest - destinated recipient id. The sender may specifies the destinated recipient of the message. However, the message is not guaranteed to be received by that process. It will only be received only if the recipient listen to the port and read the messages queued there.

key - message key for message mapping. The key allows the sender and the recipient to build an agreed message system on their own. The recipient may

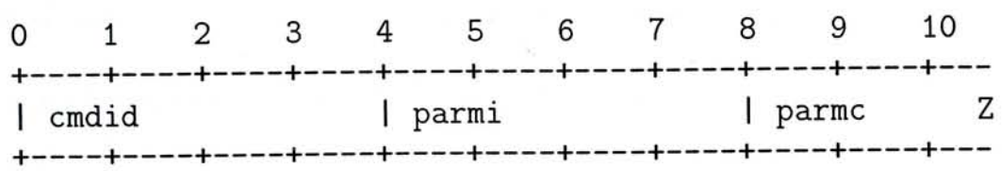
specify a key value for the message key the expect to retrieve.

A wild value ANYVALUE can be put in the key field by the sender and the recipient. the recipient may also specify a mask to specifies which bit is to be considered during the comparison of key value.

The message header is fixed at 10 bytes. The design of this compact message header is to reduce the overhead in transferring messages. The saving will be very significant especially when the message content is small.

3.5 Message Content - Control Message

The message content of data message is free structure. There is a predefined structure for control message as follow:



cmdid - command id. Each of the system service command has an id number. The user has to specifies the command id so as to invoke the command.

parmi - integer parameter. This field provide an optional integer parameter for the commands. It is not necessary for the field to be filled. A command may leave the field undefined.

parmc - string parameter. This field provide an optional character string field for the commands. It is not necessary for the field to be filled. A command may leave the field undefined. The length of the parmc field is calculated by (value of len field in message header minus length of cmdid and parmi)

cmdid - command id. Each of the system service command has an id

Any command that find the structure inefficient for its use may redefine a new structure. Users of such commands should interpret the message content as data content. They should fill the content using the new structure.

3.6 Message Transfer Functions

Message are transferred only when the process is connected to the system. The control connection is used for the transfer of commands, which describe the functions to be performed, and the replies to these commands. Several commands are concerned with processes creation and initialization. Some commands are concerned with process synchronization and group operations. A number of control information are stored in every processes for communication purpose:

mesgi - message structure contains the least received message from the system.

mesgo - message structure contains the message ready to be submitted to the system.

padd - id of parent process. This is an optional field. If a process is not started by through control-level protocol, that process will have no parent and the field is left undefined.

myadd - id of the current process itself.

cup_init

`C Synopsis - void cup_init(void)`

The routine enrolls the calling process into CUP and gets a unique system id for this process. cup_init should only be called once in each process. On successful, the fields myadd, mesgi and mesgo will be initialized to a proper value.

cup_startproc

`C Synopsis - void cup_startproc(char * pname, ProcAdd * childadd)`

The routine starts an instance of the program stated in the parameter. A successful calling of the routine will will return with a valid system id being put in childadd. The new task may be started in anywhere within the cup environment.

cup_exit

`C Synopsis - void cup_exit(void)`

The routine informs the cup environment that the calling process is leaving CUP. This routine does not kill the process. The calling process can continue to perform tasks without the support the cup environment.

cup_initsend

`C Synopsis - int cup_initsend(void)`

The routine clears the message buffer mesgo. The cleared buffer is ready for packing a new message.

cup_send

`C Synopsis - int cup_send(ProcAdd recipient, int type)`

The routine submits the message in message buffer mesgo to the cup environment. A successful return of the function indicates that the message is submitted to the environment. However, there is no indication that the recipient has received the message. The wild value ANYVALUE can be put in recipient. type is the key of the message for comparison purpose.

cup_recv

`C Synopsis - int cup_recv(ProcAdd sender, int type, int mask)`

The routine tries to get a message from the sender with type and mask matched. If there is no matched message, the calling process will be blocked until a suitable message comes. A successful return of the function will update the message buffer mesgi with the retrieved message. The wild value ANYVALUE can be put in sender. type is the key of the message and the mask indicates the bits that are effective in comparison.

cup_multicast

`C Synopsis - int cup_mcast(ProcAdd * recids, int numrecid, int type)`

The routine sends message in message buffer mesgo to all recipients listed in recips. The number of recipients is stored in numrecip and the key of the message is stored in type. The recipient uses cup_recv to receive the message.

cup_barrier

```
C Synopsis - int cup_barrier (ProcAdd * bars, int numbar, int btype)
```

The routine synchronizes a number of processes listed in bars. The processes take part will stop for a period at the same time and resume. Upon return of the routine, the synchronization is performed.

cup_joingroup

```
C Synopsis - int cup_joingroup(char * groupname)
```

If a group with groupname already exists, the calling process will join it. If the group does not yet exist, a new group named groupname will be created and the calling process will join it. The routine will return the number of processes in the group including the calling process.

cup_lvgroup

```
C Synopsis - int cup_lvgroup(char * groupname)
```

The calling process will leave the group with groupname. The routine will return the number of remaining processes in the group after the calling process left it.

cup_gbcast

```
C Synopsis - int cup_gbcast(char * groupname, int btype)
```

The calling routine will send the message in mesgo to all the process in the group

named groupname. The type of the message is btype. The calling process will also receive a copy of the message.

cup_(u)pk*

`C Synopsis - int cup_(u)pk`T`(`T` value)`

$T \in$ {int, short, long, char, float, double, ProcAdd}. This is a set of routines for data types packing and unpacking. Each packing routine has a corresponding unpacking routine. The packing routine automatically updates the mesgo and the unpacking routine automatically retrieves message from mesgi.

cup_pkbyte

```
C Synopsis - int cup_(u)pkbyte(byte * bpointer, int numbyte)
```

This routine packs numbyte pointed by the bpointer into the message buffer mesgo.

cup_upkbyte

```
C Synopsis - int cup_upkbyte(byte * bpointer, int * numbyte)
```

This routine unpacks numbyte pointed by the bpointer into the message buffer mesgo. The number of bytes stored in the number variable pointed by numbyte.

3.7 Application Development

Application programs use CUP as a general and flexible computing resource that supports message-passing model of parallel computation. This resource provides a high level *transparency* to the application programs. The underlying machine

architecture is completely hidden from the applications. Programs developed using CUP are architecture independent. Such design frees the application developer from having to take care of properties of different architecture. Exploiting the strength of a particular architecture is the responsibility of people who implement CUP on that platform.

One of the most widely used parallel programming model is the master-slave model. Master-slave model provide a simple and general model for parallel application development. In the model, there is a master program. The role of the master program is to split a job, that could be parallelize, in to n pieces. Then the master program distribute the n jobs to the m slaves, for $m \leq n$. In cases where $m > n$, it automatically reduces to the case where $m = n$ and there will be $m - n$ idle slaves. The job of slave program is to perform the computation. The partial result computed by the slaves will be return to the master. The master have to combine the partial results into a complete result. A program structure for the simplest form of master-slave is shown in list 1. In the program structure, there are n jobs and m slaves where $n = m$.

```
/* Master Program */
#include "cup.h"

#define NUMSLAVE 10
#define DATA  20
#define RESULT   30

main()
{
  int     i;
  ProcAdd childadd[NUMSLAVE];
```

```
  cup_init();
  for (i = 0; i < NUMSLAVE; i ++)
    cup_startproc("hello",&childadd[i]);

  for (i = 0; i < NUMSLAVE; i ++){
    cup_initsend ();
    /* split job and pack data */
    cup_send(childadd[i],DATA);
  }

  for (i = 0; i < NUMSLAVE; i ++){
    cup_recv(childadd[i],RESULT);
    /* unpack result and concate final result */
  }
  cup_exit();
}

/* Slave Program */
#include "cup.h"
#include "cup.h"

#define DATA  20
#define RESULT   30

main() {

  cup_init();
  cup_recv(padd, DATA);
  /* unpack data and calculate partial result */
  cup_initsend ();
  /* pack partial result */
  cup_send(padd,RESULT);

  cup_exit();
}
```

List 1. - program structure for master-slave model where n = m

In many cases, the number of jobs is very large and the size of job, say the number of instructions required to compute the partial result, is very tiny. Under these condition, creating exactly the number of slaves that equals the number of jobs and

each slave serve one job is very costly. It is because the creation of process is known to be costy and the life span of these slaves is very short. Hence the overhead in creating process could be much greater than the actual cost in computation. A enhanced version of master-slave model is shown in list 2. In the listing, the number of jobs is greater than than the number of slaves. A job queue is created and the jobs will be distributed to the slave one by one. In some design, each the jobs are evenly assigned to the slaves. Each slave will be assigned $\lceil n/m \rceil$ or $\lfloor n/m \rfloor$ jobs. In our approach, the slave after finishing a job will be assigned a new job immediately until all the jobs are completed. The merit of this approach is to keep all the slave busy instead of having some slaves have a lengthy queue of jobs while some others are left idle.

```
/* Master Program */
#include "cup.h"

#define NUMSLAVE 10
#define NUMJOB   500
#define DATA  20
#define RESULT   30

main()
{
  int     n, m;
  ProcAdd childadd[NUMSLAVE];
  ProcAdd thischild;
  Job     job [NUMJOB];

  cup_init();
  for (n = 0; n < NUMSLAVE; n ++)
    cup_startproc("hello",&childadd[i]);

  for (m = 0; m < NUMJOB; m ++){
    /* split task into NUMJOB jobs and put into job[m] */
  }
```

```
  /* send one job for each slave */
  for (n = 0; n < NUMSLAVE; n ++){
    cup_initsend ();
    /* pack data for job[n] */
    cup_send(childadd[n],DATA);
  }

  /* send job to the slave that is free */
  for (m = NUMSLAVE; m < NUMJOB; m ++) {
    cup_recv(ANYVALUE,RESULT);
    cup_upkProcAdd (&thischild);
    /* unpack result and concate final result */
    cup_initsend ();
    /* pack data for job[m] */
    cup_send(thischild,DATA);
  }

  /* get remaining result from slave and terminal slaves */
  for (n = 0; n < NUMSLAVE; n ++){
    cup_recv(ANDVALUE,RESULT);
    cup_upkProcAdd (&thischild);
    /* unpack result and concate final result */
    cup_initsend ();
    /* send empty message */
    cup_send(childadd[n],DATA);
  }

  cup_exit();
}

/* Slave Program */
#include "cup.h"

#define DATA  20
#define RESULT   30

main() {

  cup_init();
  for (;;) {
    cup_recv(padd, DATA);
    if (length of DATA == 0) break;
    /* unpack data and calculate partial result */
```

```
    cup_initsend ();
    /* pack partial result */
    cup_pkProcAdd (myadd);
    cup_send(padd,RESULT);
  }
  cup_exit();
}
```

List 2. - program structure for master-slave model where n => m

The program constructs of CUP provides a set of flexible tools for the parallel application developers. There are no limitations to the programming paradigm a CUP user may choose. Different models for parallel programming may also be applied using the CUP message passing resource. These models include pipeline, single program multiple machine (SPMD), multiple program multiple machine (MPMD) and etc. The choice of a proper programming model is up to the experience and knowledge of the program developer.

Chapter 4

Multiple Computer Infrastructure

The project is to develop a *parallel processing architecture* based on Pentium PC and SCSI bus. A low cost parallel machine, that could be used to implement *fine* to *coarse* grain algorithms, is built. The current implementation is essentially a message passing based multiple computer system.

The complete architecture consists of three layers: *Application Support*, *Local Message Routing*, *Network Message Routing*. The implementation of the three layers supports the functions in the specification of our new protocol namely CUP. The infrastructure and the implementation of the system is code named as *IRI*. The current implementation is based on the existence of the *SMQ* and *SCSI hardware and device driver*. The relationship of the entities is depicted in fig 4.1. The Application Support Layer provide an interface and environment to the parallel applications using the IRI system. The Local Message Routing Layer is responsible for distributing messages with the same host. The Network Message Routing Layer

is responsible for getting messages from the Local Message Routing System and distribute the network messages to the Local Message Routing System on the correct host. Once a network message arrives the Network Message Routing Layer for the correct host, it will be passed to the Local Message Routing Layer.

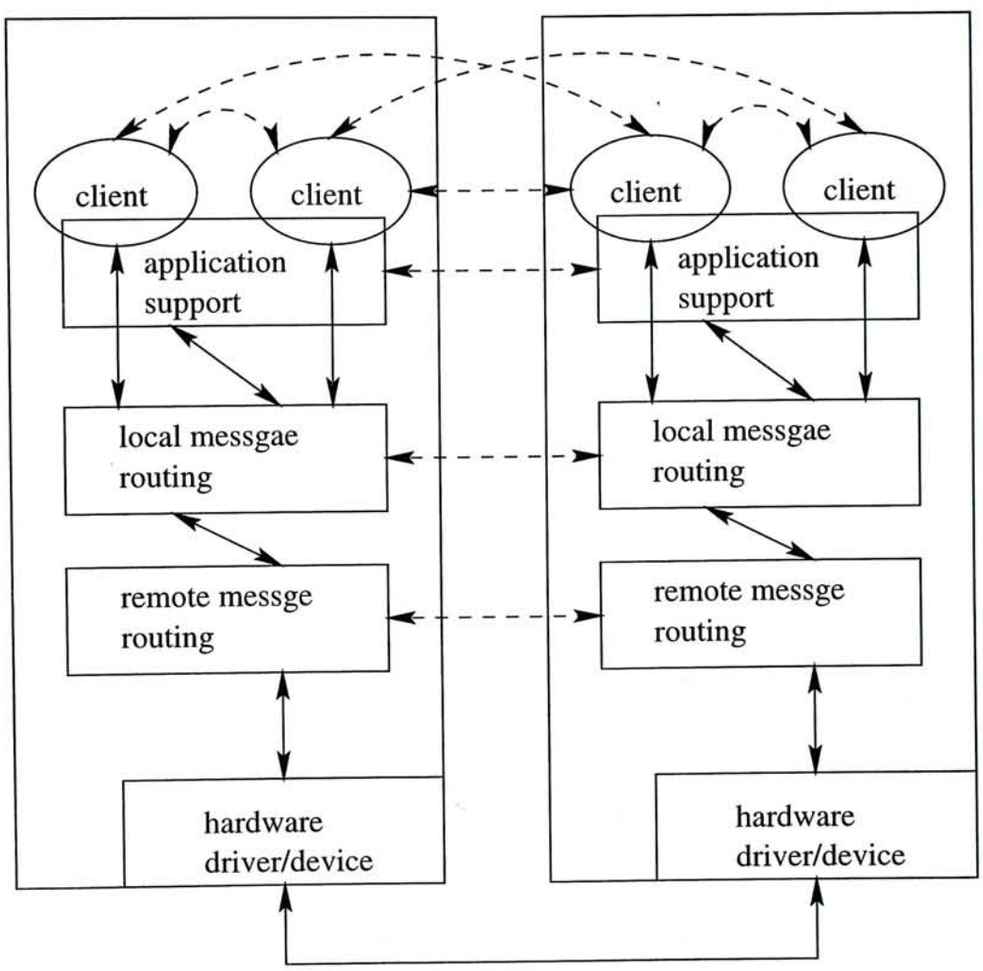


Figure 4.1: Architecture of IRI system

Our work is performed on the following platform:

Intel based PC -

Intel based PC is chosen as the basic build block of our system for it is economic and popular. The sole use of Intel as our build block of the IRI system

restricts the resulting system to be an homogeneous system. It is unusual to build a multiple computer basing on homogeneous machines. It is because networks usually consists of a wide variety of machines from different vendors. It is believed that building a homogeneous system will under utilize the computing power of a *network of workstations (NOW)*. On the other hand, building an heterogeneous system, the difference in representation of data structure has to be take care. This includes the length of different data structures, and the order of bit and byte. These difference in representation would induce overhead in communication. Since the IRI system is to support high volume of small messages, lower overhead in communication is preferred. The existing system consists of four identical Pentium machines. The computing power of the machines are of the same level. Identical machines are used because the system is to support highly coupled application. Studies shown that, running highly coupled parallel applications on a multiple computer with different machines, the performance will degrade to the level of the slowest processing unit.

Adaptec 2940w SCSI Adapter -

At the very beginning of the project, the Adaptec 2940w SCSI adapter is the only available SCSI adapter that supports FAST-WIDE SCSI protocol [Aic95, ANS94, Sch95]. The adapter supports 10 MHz and 16 bit wide data transfer. The Adaptec 2940w is a single chip host adapter. The chip on the adapter is AIC-7870 which provides advanced host adapter features with

SCSI-2 bus controller and a full featured PCI 32-bit bust master. The AIC-7870 incorporates a dedicated processor, the SCSI Phase Engine, which is a RISC sequencer. Scratch RAM on the adapter allows much of the SCSI instructions to be off loaded from the kernel. The Sequencer runs at 10 MHz. It phrase the SCSI instructions and operates the SCSI BUS. As a result, the host CUP is free from handling the most SCSI operations. In our implementation, the host adapter automatically handles all SCSI phases in a initiator mode. In target mode, the only job done by the kernel is to allocate memory space for storing the incoming data. The kernel is drives the sequencer through direct register accesses. The sequencer informs the kernel of events through PC Interrupt system.

FreeBSD 2.0.5 Operating System -

The FreeBSD Operating System is a variant of the 4.4 BSD-Lite Operating System from the University of California at Berkeley [LeM90]. The operating system is stable and well documented. Since much of the work has to be done within the kernel, the stability and documentation of the OS becomes an very important in the choice of OS. It provides a preemptive multitasking and multiuser environment for its user. The entire source code of the operating system is freely available. Kernel facilities like the SMQ and device drivers could be added when needed [Paj91]. Besides, FreeBSD is a very popular platform that many other parallel environment tools are official ported and supported. So that fair comparison with other packages can be made.

The above platform could easily be built using off-the-shelf hardware. It is easy to upgrade and expand.

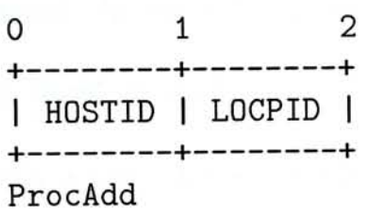
4.1 Application Supper

The *Application Support Layer (ASL)* provides a *transparent environment* to the tools and applications running on the message routing system. Clients attached to this layer are capable of sending and receiving messages through the system. They are also served by a number of *system processes.* The lower layers of the system provide essentially only simple message sending and receiving functions. The operations specified in our *CommUnication Protocol (CUP)* are implemented as a function library (*CUPLIB*) with which clients have to be linked. In other words, CUP describes what functions have to be supported and CUPLIB is how the functions are supported. It supports the functions stated in CUP on our hardware and operating environment. The latest implementation is based on the knowledge of the underlying platform and optimized for this platform. *SMQ system & SCSI device* is used in passing messages. The implementation is system dependent and is not portable to the other platforms unless the corresponding SMQ & SCSI interfaces are also ported.

Each client attached to the system is assigned an *unique id*, which is also used its *communication port id.* Client processes use the library calls to send and receive messages following the message formats specified. The *command handler* is a

system process that serve the clients. It is also implemented using the CUPLIB function library. It carries out special requests from other clients on both the same host and the remote hosts. It is also responsible for maintaining and monitoring the operation of the environment. Example of requests handled by the command handler are creation of child process (local/remote), multicast and etc.

For easy implementation, the ProcAdd structure is subdivided into two fields in the CUPLIB: HOSTID and LOCPID.



HOSTID - Host id on which a process resides.

LOCPID - Local process id of a process on the local host.

The HOSTID in the current implementation is an id assigned to each of hosts on the IRI system. The LOCPID is a unique id of a client process within the local IRI environment. This structure is limits the number of hosts of to be 255 and number of client on each of the hosts be 255. In addition, it prohibit the investigation of other problems such as *process migration strategies* using the system. However, the implementation of the other parts of the system is much simplified. Besides, the aim of the project is to build a parallel system using a high speed data link and low cost machines. The limitations introduced does not affect our studies.

A copy of the command server is located in each of host machines. The system id of command server is 0x?1. All the system processes are assigned fixed system ids. In the current implementation, the well 0x?[1..10] are reserved for system processes and are known as well known system ids. All the client of the system could assume that the system ids are statically assigned to fixed system processes. Therefore the client may communicate with them and request for services.

The CUP does not specify any implementation detail of the functions to be supported. In the CUPLIB, the detail of the functions is described. These functions are implemented using the basic send and receive facilities. The CUPLIB is essentially a series of *communication sequence* between client-client and client-command server for achieving the tasks. The sequences is predefined and agreed and the actual sequence is hidden from the clients.

The CUPLIB is still under development. The group operations specified in the CUP is implemented by another group of people and will not be described here.

4.1.1 Send and Receive

The application support layer provides facilities for asynchronous message passing. The sender after sending the message, will return to work. It makes no assumption on when the recipient-could-be will get the message. A task may issue a receive command ignoring whether there is message in the message queue. However, it will be blocked until its message queue is non-empty (Figure 4.2,4.3). The users may

use the asynchronous message passing facilities to support synchronous message passing if it is required. Send and receive are primitive commands in the library.

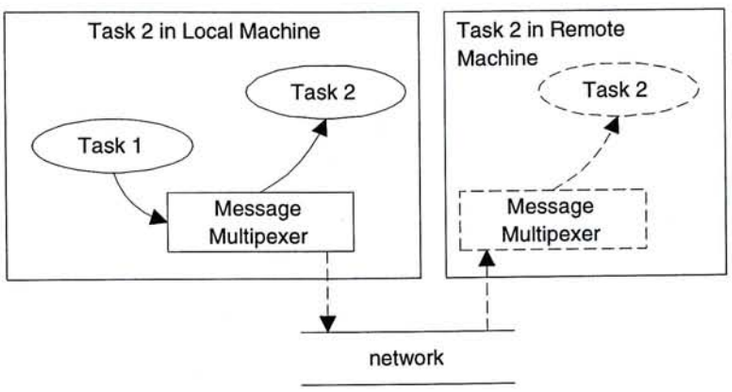


Figure 4.2: Sending Message to Local/Remote Receiver

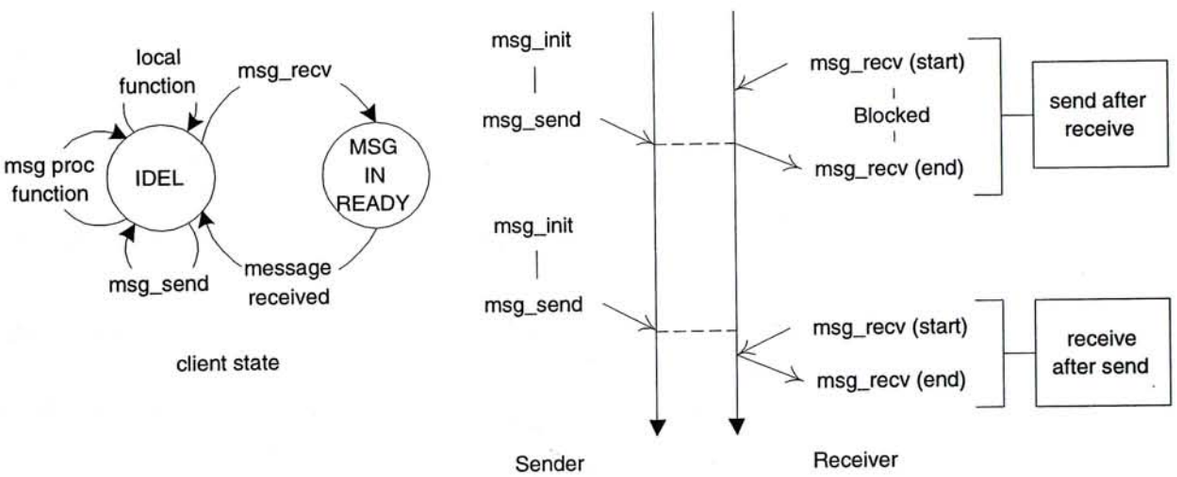


Figure 4.3: State Diagram of Client Process and Timeline of Client

4.1.2 Multicast

Multicast allows a sender to send a message to a group of receivers. The group of receivers may or may not include the sender. The multicast command is implemented using the send command.

```
multicast
  input rec_list
```

```
begin
  for all rec in rec_list
    msg_send (rec)
end
```

As from the above pseudo code, the implementation of the multicast is very simple. The sender passes a list of recipients to the routine. The routine sends the message to each of the recipients. The recipient uses the recv routine to receive the message.

4.1.3 Barrier Synchronization

Many problems can be solved by using algorithms that iteratively compute better approximations to an answer, until an acceptable solution is arrived. Multiple processes may be used to look after disjointed partial solutions. The solution of stage n must be completed before stage $n + 1$ may started. In other words, all the processes must be synchronized before starting the next stage. This form of synchronization is known as barrier synchronization. To perform barrier synchronization, all the processes involved have to arrive the barrier before any may pass (figure 4.4). The barrier function is implemented using the multicast, send and receive commands.

```
barrier
  input barrier_list
begin
  if i am first member in barrier_list
    for all barrier_member in barrier_list
      wait for barrier message
    init barrier end message
    multicast(barrier_list except me)
```

```
  else
    init barrier message
    msg_send (barrier message)
    msg_receive (barrier end message)
  endif
end
```

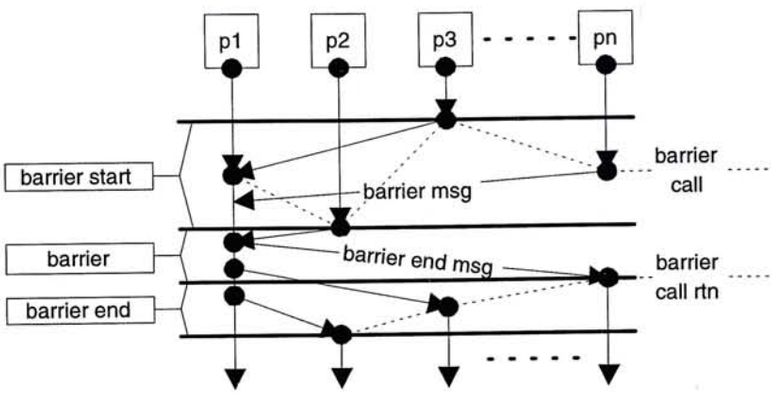


Figure 4.4: Barrier Synchronization of a number of processes

4.1.4 Start and Delete Process

A task may instruct the command server to start a child task and subsequently delete it. The start function will cause the caller to block until a reply containing the id of the new process is returned from the command server is received. After issusing the delete instruction, the caller may assume that the commander will finish the task properly and it will not be blocked (figure 4.5). Both the start and delete process are implemented using send and receive commands.

```
Client
startproc
  input proc name
  output proc id
begin
  init command message
  msg_send(to command server)
```

```
deleteproc
  input proc id
begin
  init command message
  msg_send(to command server)
end
```

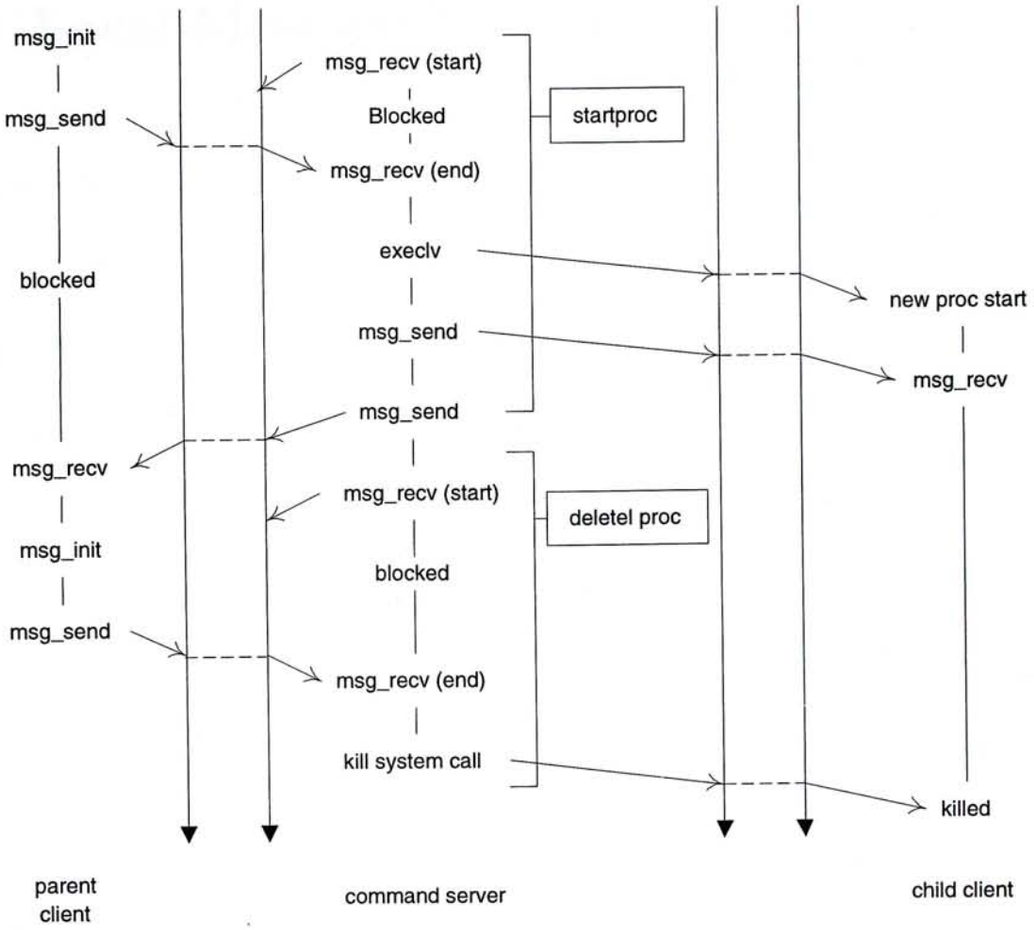


Figure 4.5: Action of startproc and deleteproc

```
  msg_recv(from command server)
  extract proc id from message
end

Command Server
begin
  while receive command message
  begin
    extract parameter from message
    if command startproc
       start new proc to execlv(proc name)
       send parent id and proc id to new proc
       send proc id to parent
    if command deleteproc
       remove proc id
  end
end
```

4.2 Local Message Routing

The *Local Message Routing Layer (LMRL)* accepts submission and request of messages from the clients. It also passes and gets network messages from the *Network Message Routing Layer (NMRL)*. The LMRL defines a set of mechanism for enveloping and un-enveloping a CUP message into the format supported by the IPC facilities used. In general, any IPC facilities can be used to support local message routing. In our project, *Berkeley socket* and *System V Message Queue* have ever been used in routing local messages. In the latest system, *SMQ* was developed for supporting the Local Message Passing Layer. With respect to the Local Message Routing layer, all the messages from clients in any remote host are of no difference. They are collectively known as network messages. There is no difference between a message to *host A* and a message to *host B*. There is also no difference between messages to *client C1*, *client C2* on *host A* and *client C3* on *host B*. However, the Local Message Routing Layer does differentiate messages to different clients on the local host (fig 4.6.) There is a 1:n (Network:Local Clients) relationship according to the LMRL.

Using different IPC, the implementation of the LMRL will be very different. In the following sections, the implementation using the three different IPCs: Berkeley socket, System V Message Queue and SMQ will be discussed. In each implementation, two interfaces: cup_send and cup_recv are provided to the application support layer. The applications have to link with the library containing the interface. The

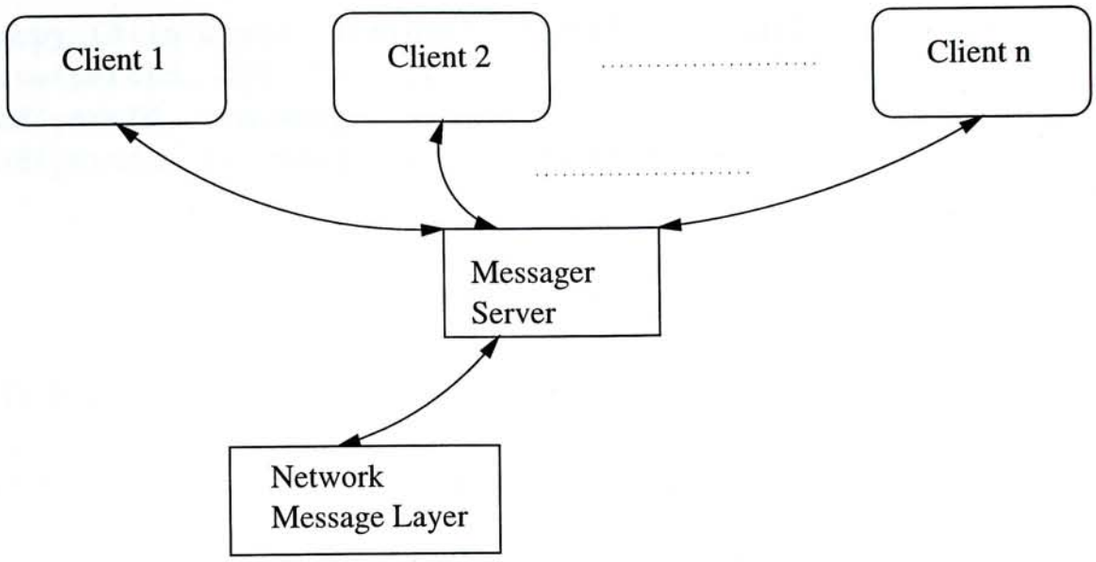


Figure 4.6: Local Message Routing Layer

internal implementation of the LMRL is hidden from the upper layer.

4.2.1 Berkeley Socket

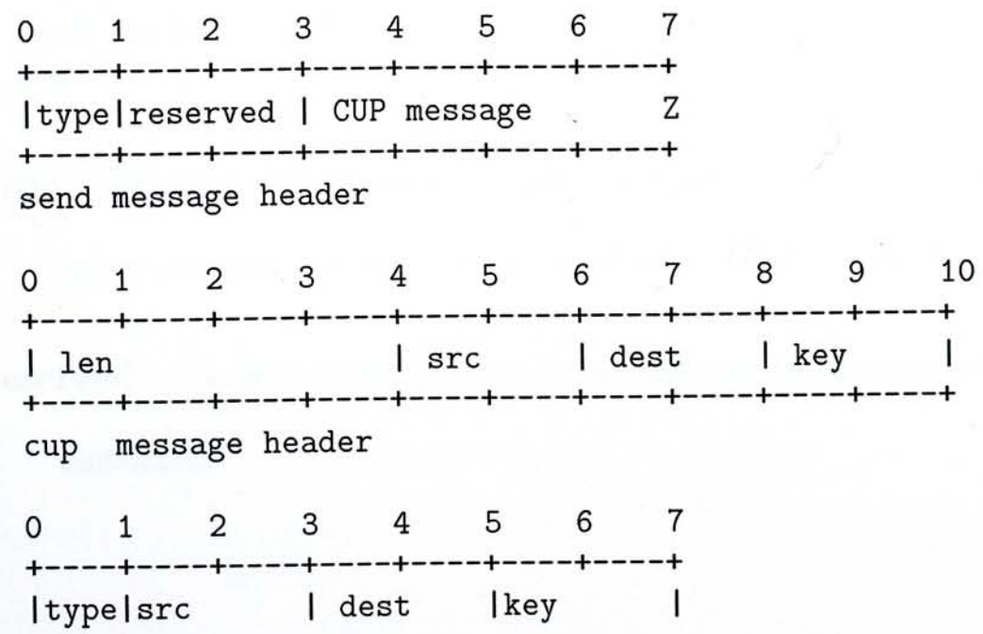
TCP socket was first IPC used in implementing the LMRL. In this implementation, each of the clients has a TCP socket for sending and receiving messages. The following statements are used in read and writing to a socket:

```
int cup_send(ProcAdd todest, short int outtype) {
...
  out.type = SEND;
  out.mesgo.key = outtype;
  memcpy (&(mesgo.src), &(myadd), sizeof (ProcAdd) );
  memcpy (&(mesgo.dest), &(todest), sizeof(ProcAdd));
  write(portfd, tout, out.mesgo. len + 3);
...
}

int cup_recv(ProcAdd frsrc, short int intype) {
...
  in.type = RECV;
  in.key = intype;
  memcpy (&(in.src), &(frsrc), sizeof(ProcAdd));
```

```
  memcpy (&(in.dest), &(myadd), sizeof(ProcAdd));
  write(portfd, &in, 7);
  read(portfd, tin.mesgi, 4);
  read(portfd, tin.mesgi.src, in.mesgi.len);
...
}
```

The TCP socket is not a *selective* but the LMRL has to support selective message retrieval. In our design, a local *message server* collects all the message and a *message system* was designed for submitting and retrieving messages. The message server holds a table that maps a socket port to a CUP communication port. It also maintains a list of messages to be retrieved by local clients. A list of messages requested by the clients is also maintained (fig 4.7) Clients submit two types of messages to the message center: 1. send message & 2. request message. The format of the messages are shown below. The simple message system between the client and the message center is applied to control the flow of application level messages.



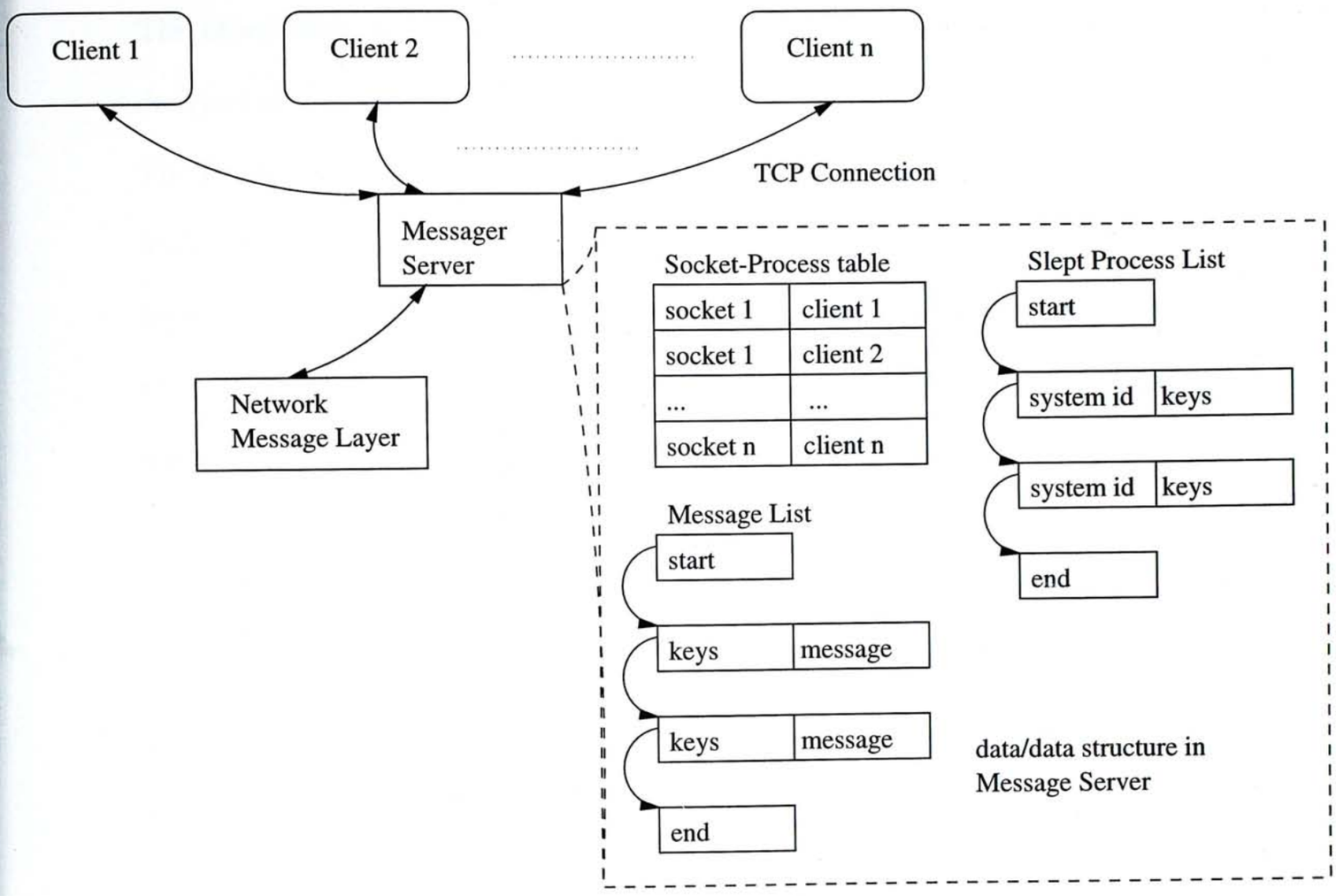


Figure 4.7: TCP Socket Implementation of Local Message Routing Layer

```
+----+----+----+----+----+----+----+
send message header
```

type - the type of the message. This is a control field the informs the message server program whether the clients want to send or receive a message.

reserved - this field is reserved for future use and the content of the message is undefined.

The other fields are of the same definition as in CUP. The message format is designed to minimize the number of read operations required over the socket port. The length of send message header is 3 bytes. The length of the receive message header is 7 byte. Hence, the length of a send message header together with the length field of a CUP message equals that of a receive message. This design requires two socket reading for send message and one for receive message. In the first read, the message server reads 7 bytes. If it is a receive message, the operation is complete. If it is a send message, the length field of the CUP message is used to calculate the number of remaining bytes to be read. The interaction sequence between the client and the message server for sending and receiving messages is depicted in fig 4.8. In the figure, Case I is the situation that when the recipient request for message, the sender has already submitted it. Case II is the situation that the sender submits the message after the recipient request for it.

Sending a message is a comparatively simpler operation. It is a *one-pass operation.* The sender passes the message to the message center through the port. After submitting the message, the sender may consider the operation as finished and return to its work. The message server, checks if there is already a recipient waiting this message. If the answer is positive, it will send the message to that recipient. Otherwise, the message will be put into the message list.

Receiving a message is a *two-pass operation.* The recipient submits the received message to the message server than it will block and wait for the requested message.

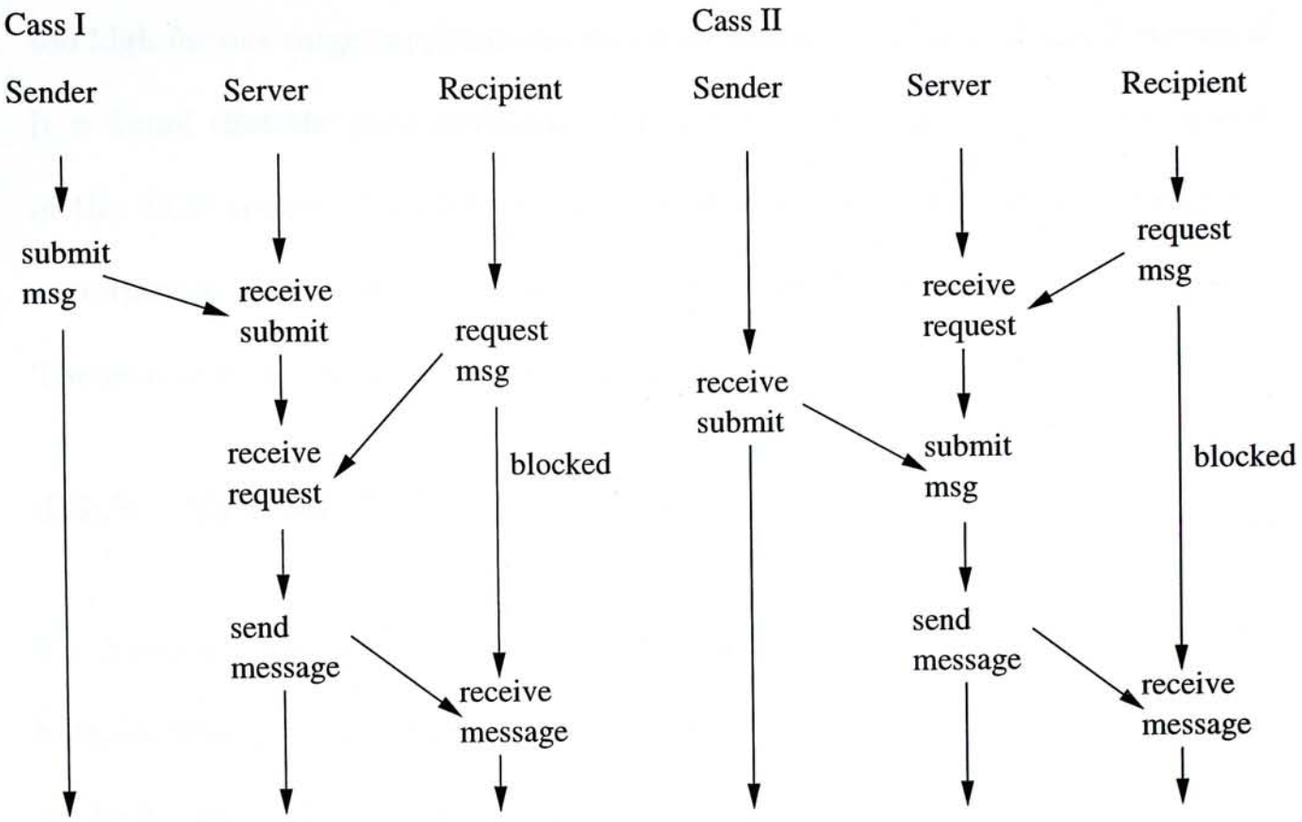


Figure 4.8: Interaction Between Message Server and Clients

The message server, after receiving that message will check if there exists any CUP message fits the requirements. If there exists such a message, the message will be removed from the message list and pass to the recipient. Otherwise, the request of the recipient will be put into the request list. The request list will be checked each time a new message is submitted from the local clients or from the Network Message Routing Layer.

The TCP socket provides a reliable mean for IPC. The message server together with the TCP socket provides a reliable and selective message passing system. However, when performance is an concern, the TCP socket implementation does not provides a satisfactory result. The performance of local passing routing is of the order of the PVM. The overhead of passing message is considered to be

too high for our target applications which creates large volume of small messages. It is found that the poor performance is resulted from the internal performance of the TCP socket. In addition, the use of a message server also degrades the performance. It is because, a task switch is resulted in each local message pass. The cost of task switching is also part of the communication cost.

4.2.2 System V Message Queue

The System V Message Queue is an IPC that limits to a support the communication between processes on the same host. In its implementation, all messages are stored in the kernel. It is possible for a process to put a message into the queue and have the message get by another process at a later time. In contrast with TCP socket implementation in which every client has to make a connection to the message server, all clients can read message and write to a single multiplexed message queue. The System V Message Queue supports key matching in message retrieving and the key can be used to multiplex the queue.

```
int cup_send(ProcAdd todest, short int outtype) {
...
  mesgo.mesg_type = myadd | (todest<<sizeof (ProcAdd) );
  mesg_send(que.idwrite, &mesgo);
...
}

int cup_recv(ProcAdd frsrc, short int intype) {
...
  mesgi.mesg_type = frsrc | (myadd<<sizeof (ProcAdd) );
  mesg_recv (que.idread, &mesgi);
...
}
```

The Message Queue supports a 4 byte length key matching. To simplify the operation, the CUP specified support for type matching is not included in the this implementation. The *src* and *dest* field is packed into the 4 byte key for message matching. A message server plays a different role than that in the TCP implementation. In the TCP implementation, all the messages are stored in the message server. The message server is also responsible for matching the key submitted by clients and that of messages, and, pass messages to recipients. In the System V Message Queue implementation, the role of message server is in fact a *multiplexer.* The implementation is depicted in fig 4.9. The only work of the message server is to move message from incoming queue to outgoing queue. It also has to pass the network messages to the lower layer. However, the Network Message Routing Layer simply puts the input messages from the network into the input queue for the sake of performance. It is to reduce the work or message server and the frequence task switching. The direction is with respect to the clients. All the messages are stored in the kernel and the System V Message Queue facility is responsible for the job of key matching.

The use of System V Message Queue is an partial implementation. It is used to evaluate the performance difference between different IPCs. It boost up the performance of local message routing by more than 3 times of the TCP implementation. This implementation is considered satisfactory. Though some problems observed in the TCP implementation are still applied. The problems are: *process switching* for each message passing, *unnecessary memory copy operations.* Detail studies of

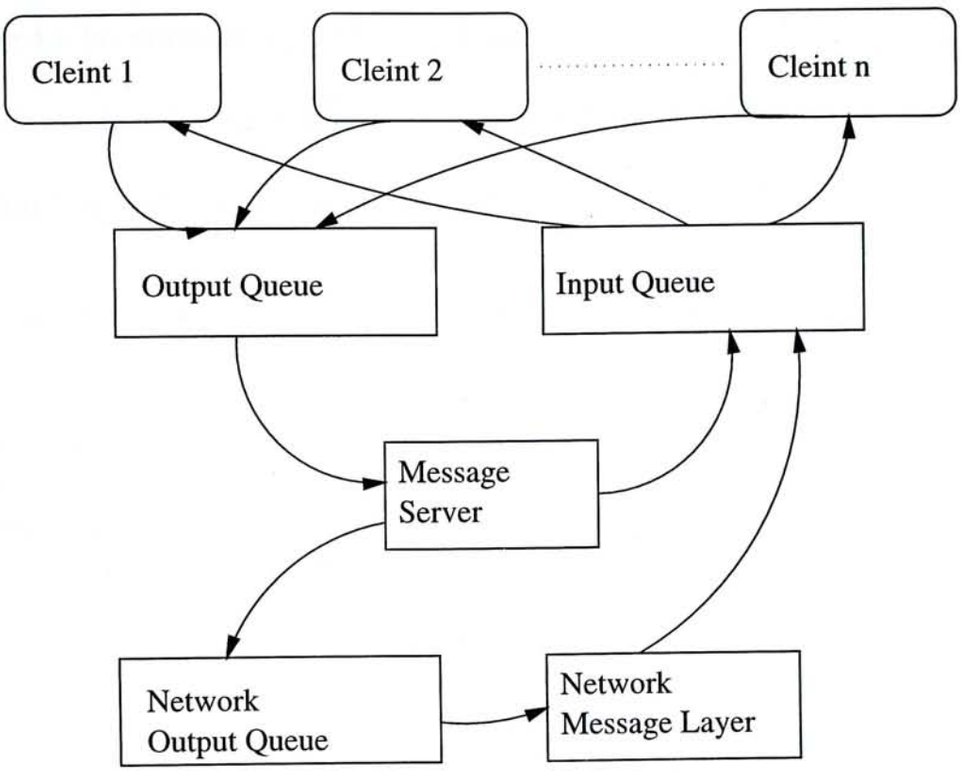


Figure 4.9: Message Queue Implementation of Local Message Routing Layer

the System V Message Queue shown that better performance could be resulted by avoiding un-necessary memory copy operations. In the System V Message Queue, there are at least 2 memory copy operations occurred for each message passing. One copy is occurred when the sender puts the message content into the Queue. The other copy is occurred when the message is moved to the recipient's message buffer.

4.2.3 Shared Memory Queue SMQ

The design of the Shared Memory Queue is in fact a combination of the System V Message Queue and the TCP Socket implementation. In also make use *shared memory* to improve the overall performance. The design of the SMQ to makes

it suitable to be connected with the Application Support Layer. It provides a number of interface for flexibility and the interface provided by the LMRL to the Application Support Layer are as follows:

```
int cup_send(ProcAdd todest, short int outtype) {
...
  mesgo.type = outtype;
  memcpy (&(mesgo.src), t(myadd), sizeof(ProcAdd));
  memcpy (&(mesgo.dest), t(todest), sizeof(ProcAdd));
  if (mesgo.dest.bus0 != mybus0) mesgo.type=(0x8000 1 outtype);
  smqenque (mesgo);
  smqloc (&mesgo);
...
}

int cup_recv(ProcAdd frsrc, short int intype) {
...
  smqdeloc (mesgi);
  mesgi.type = intype;
  memcpy (&(mesgi . src), &(frsrc), sizeof (ProcAdd) );
  memcpy (&(mesgi . dest), t(myadd), sizeof (ProcAdd) );
  smqdeque (&mesgi, 0xFFFF);
...
}
```

The main component of the LMRL layer is the SMQ facility. The SMQ key fields provide a direct mapping to the CUP *srd*, *dest* and *key* fields. The architecture of the implementation using SMQ is depicted in fig 4.10. There's no *message server* in the SMQ implementation. All the comparison are performed with the kernel and minimum task switching is required. The *most significant bit* of the key field in the CUP message is used to identify whether the message is a *local* or *network* message. When the client submits a message, the ASL sets the bit properly. The Network Message Routing Subsystem is also attached to the SMQ. It will use the

mask to get all the network messages from the SMQ. It also put the incoming network messages into the SMQ for retrieving by local clients. Before the message are put, the bit is reset by the NMRL.

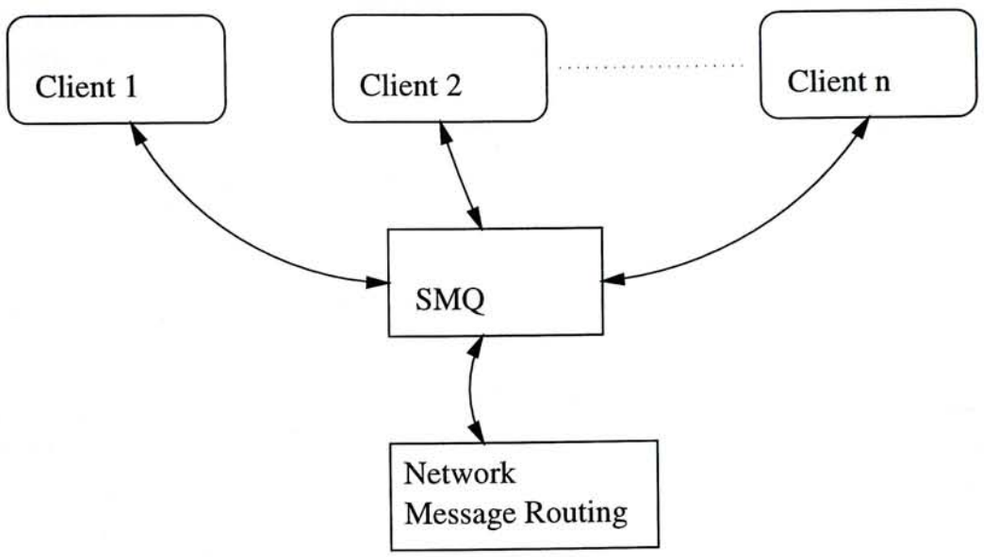


Figure 4.10: SMQ Implementation of Local Message Routing Layer

4.3 Network Message Routing

The *Network Message Routing Layer (LMRL)* is essentially an *multiplexer* that routes messages between its *local host* and a *network of hosts* (1:n). It only distinguish messages to and from different hosts. However, it does not distinguish messages to Client A or Client B on any host. It provides a transparency to the upper layers that the environment is a network of computers. A system process, *network message router*, is implemented to distribute messages to and from LMRL on the remote hosts (fig 4.11.) Two different implementations have been developed. One uses a *10 Mbit Ethernet* and *TCP Socket*. The other uses the *SCSI Link*. The 10 Mbit/sec Ethernet implementation is used as a test bed for the upper

layers before the SCSI Link is available. The SCSI Link is the final version used in the IRI system.

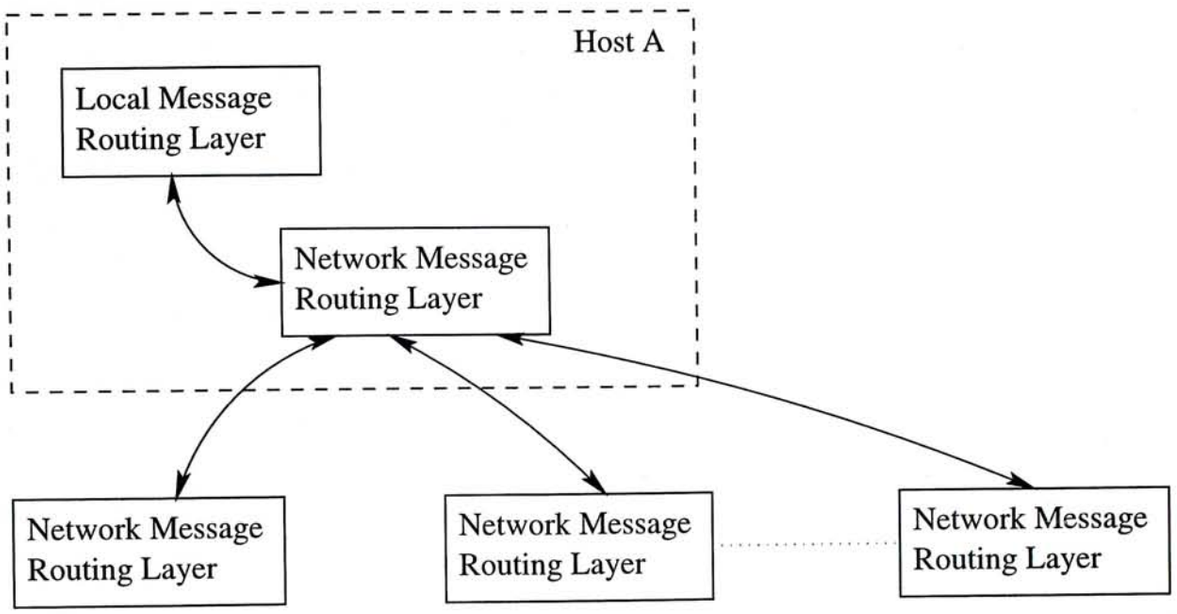


Figure 4.11: Network Message Routing Layer

The performance of the NMRL depends highly on the performances of the underlying network, the protocol of the network, the protocol that utilizes the network and the quality of the hardware device driver. The architecture of the NMRL is very different in the two implementations. There is a large gap in the performance of the different NMRL implementation. The SCSI Link provides a much faster throughput and the reason will be explained later. In the both implementations, the maximum number of hosts is limited to 15. Each of the hosts have a unique host id on the network.

4.3.1 Ethernet & TCP Socket

In the Ethernet implementation, the hosts are by to a 10 Base/2 co-axial cable. The maximum data rate of the cable is 10 Mbit/sec. The media is know as a CSMA/CD. The network runs a IEEE-802 LAN protocol. The network adapter is the 3Com Etherlink III. The device driver of the adapter card consists of about 2000 lines of source code. TCP Socket is used to connect the network message routers. All the network routers are fully connected (fig 4.12.) The figure shows the case where there are 4 hosts in the system and the hosts are fully connected.

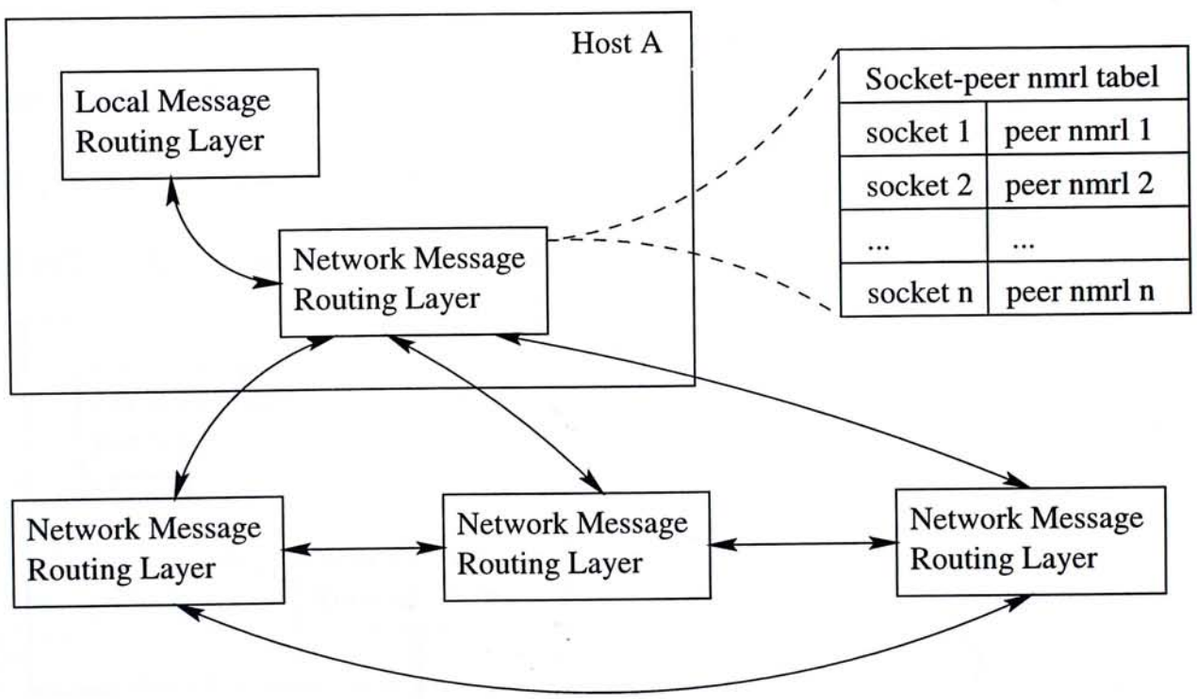


Figure 4.12: TCP Socket Implementation of Network Message Routing Layer

In this implementation, the network message router maintains a table of TCP socket and host id in the IRI system. It listens to the local message routing layer for outgoing message. When an outgoing message is received, the target host id is checked with the table to find the correct TCP socket. Then the message is

sent to that socket. The network message router also listens to all the socket that connected to the peer network message routers. Whenever a message comes, it pass it into the local message routing layer.

4.3.2 SCSI Link

In the SCSI Link implementation, all the hosts are connected to a parallel SCSI cable. The maximum throughput of the SCSI Link is 20 Mbyte/sec. The network message router directly communicate the SCSI device driver to pass messages to and from the network (fig 4.13.) In order words, there is only the SCSI protocol runs on the network. The network router is directly accessing the network. The complexity of device driver for the SCSI adapter is much higher than that of the Etherlink III. It consists of over 4000 lines of source code.

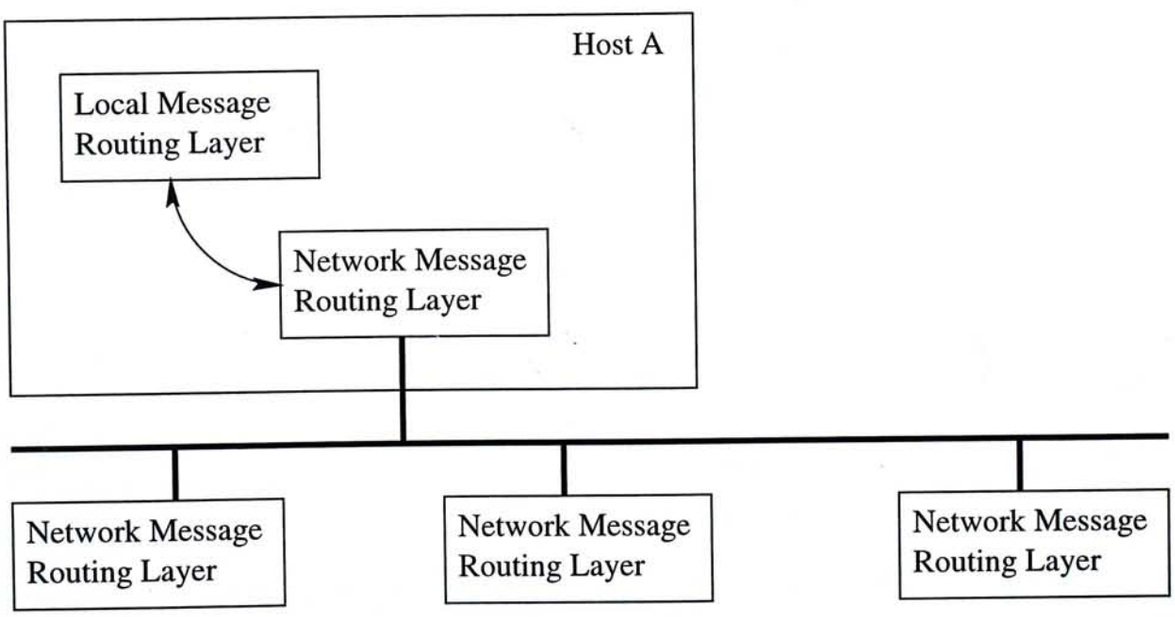


Figure 4.13: SCSI Link Implementation of Network Message Routing Layer

In this implementation, the network message router simply listens to the Local

Message Routing Layer and the SCSI device driver for messages. Messages from the LMRL is simply passed to the SCSI device driver using the send entry point provided. Similarly, messages from the SCSI device driver is passed to the LRML with processing. In order words, it behaviors as a simple conveyer belt that connects the LMRL and the SCSI device driver. The scsimsg structure is designed to overlap with a LRML layer message. Therefore, no additional processing is required by the router before passing the messages back and forth.

Chapter 5

System Supporting Facilities

Two kernel facilities are designed and developed to support the operation of the IRI system. They are the *Selective Memory Queue (SMQ)* and the *SCSI Device Driver.* They are not part of the IRI system architecture. In order words, they could be replaced by other facilities if equivalent functions and interfaces are available. The Selective Memory Queue is used in the Local Message Routing Layer. It provides a low cost interprocess communication facility for process within the same hosts. The SCSI Device Driver is used by the Network Message Routing Layer. It allows the IRI system to use the high speed SCSI Bus.

5.1 Kernel Message Support

In previous versions of IRI, *Berkeley Socket* and *System V Message Queue* are used for *local message passing.* The performance of systems using these IPC facilities are compared. Both facilities have the property that they are easy to use. We

found that message queue outperforms socket a lot. This is due to the internal simplicity of message queue as compared to socket. To pass a local message, sockets have to loop messages through the protocol layers. The majority cost spent in message queue is the memory copy operation. To pass an message using the System V Message Queue, two memory copy operations are required. One copy occurs when the message is duplicated to the message queue. The other occurs when the message is copied from the queue to the recipient.

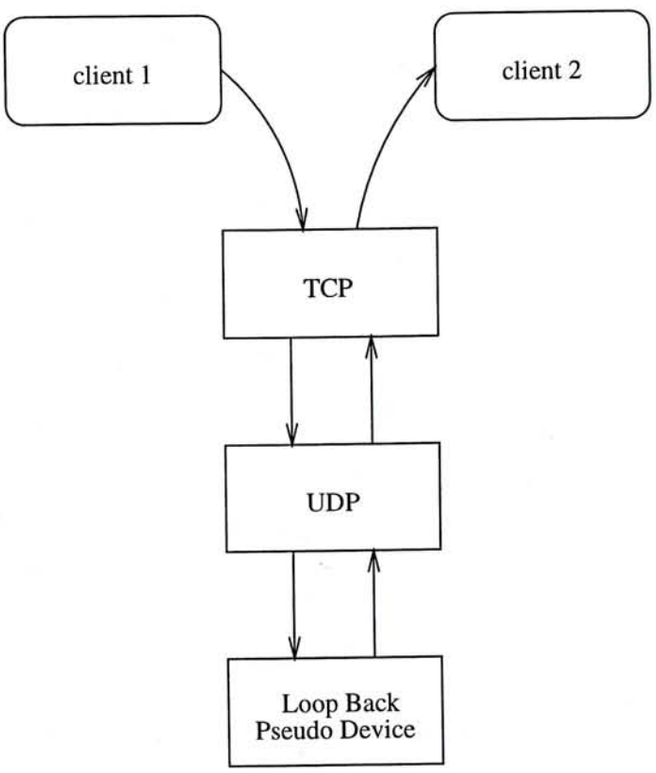


Figure 5.1: TCP Socket Message Passing

The *Shared Memory Queue (SMQ)* is developed to further reduce the message passing cost due to memory coping. It is implemented as a system service. The internal of the SMQ is equipped with a common memory pool and two queues. The memory pool is split into *segments*. The segments are the place where actual

message contents stored. One of the queues stores the starting address, length and other attributes of memory segments. The other queue stores the processes that slept and waiting for messages. All the client processes have to attach to the memory pool for communication. Before putting a message into the queue, a process have to claim a memory slot for holding its data. After moving the data into the memory slot, it enqueue the memory slot to the kernel with the keys that identify the message. The kernel will take care of the slot until a process ask for a message with keys matched. The memory slot will then be assigned to the recipient process. It will not be released until the recipient reuse the slot and enqueue it or explicitly declaim it. A process may hold more than one memory slot at any time after attachment and before detachment.

Within the kernel, the SMQ monitors the utilization of the message queue and the memory pool. It also keeps track of the blocked processes waiting for messages. There are totally three key fields. A message is received if and only if all the keys specified by both sender and recipient could be match. If ANYKEYVALUE is put in a field, then the comparison result of that field is always true.

The SMQ system provides 6 interfaces to the user of the system.

smqinit - This system call provides a mean for the user of the system to initialize the SMQ system. Once the function is invoked, the complete SMQ will be cleared. All the queues are flushed. All the segments will be removed and return to the memory pool.

smqgid - This system call allows the user processes to attach to the SMQ system. The calling process provide a key for of the expected queue system. If the key matches with the key of the existing system, the calling process will be allowed to use SMQ. Otherwise, an error will be returned.

smqloc - This system call is used to request a memory segment from the memory pool. The size of segments available is limited to the size of the memory pool or the segment with maximum size in the pool. If the memory pool or the queue is full, an error will be returned. The case which the requested size is larger than the available size will also cause an error.

smqdeloc - This system call is used to return an allocated memory segment to the memory pool. The processes should free all the allocated memory segment before exit. Freeing and unlocated segment will return with an error status.

smqenque - The processes use this call to enqueue an message to the SMQ system. Three keys that identifies the message have to be submitted with the message. Enqueuing an message with an unallocated memory segment or with a segment not allocated to the calling process is an error.

smqdeque - The user processes use this call to request for messages. The request of an unexisting message will cause the calling process to be blocked. The slept process will be wakeup until a message that matches the request.

Each of the memory segments has three different status: FREE, ALLOCATED,

SUBMITTED. FREE segments are the segments that is not assigned to any processes. After an initialization of the SMQ system, the whole memory pool is treated as a segment with FREE status. ALLOCATED segments are those segments that are assigned to a process. They are cut from FREE segments. The owner of the ALLOCATED segments could access that segment without any limitation. The owner of an ALLOCATED segment, after putting the content into the segment, will pass it to the SMQ system. The segment will be marked as SUBMITTED. An SUBMITTED segment does not have an owner and it is waiting to be retrieved by any attached process. When an SUBMITTED is retrieved by a process, it will be marked as ALLOCATED and the retrieving process will be its owner. Garbage collection is performed when a message segment is deallocated. Small neighbor segments will be concate into bigger FREE segments. The ALLOCATED and SUBMITTED will not be moved. The memory pool is therefore a number of FREE segments separated by ALLOCATED and SUBMITTED segments. There will be no continuous FREE segments. Figure 5.2 shows the change of states of memory slot.

In System V Message Queue implementation, when a message is received, all the blocked process are wakeup and the processes will compete to check if the message fits its criteria for receiving. Besides the order of message retrieval is not guaranteed. That is a process that request the message later may win a message the suit a process that request it earlier. In our implementation of the SMQ, the module stores the criteria specified by each of the slept processes. Both slept process infor-

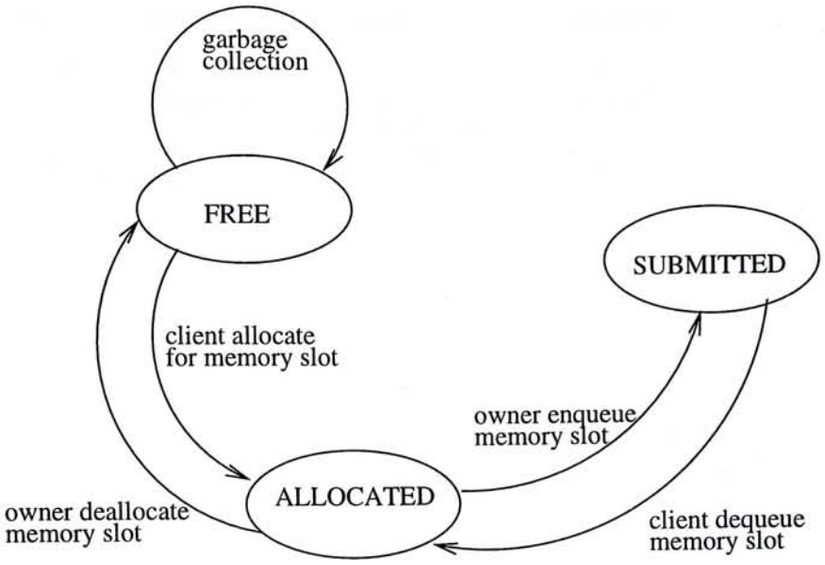


Figure 5.2: TCP Socket Message Passing

mation and the message retrieval criteria are stored in the order of arrival. When a message is received, the module check to find one process to be wakeup following the order. Therefore, When there are more then one processes waiting for such a message, only the first arrival process will be wakeup. It will then remove the message from the queue. Since only one process is waken up, the costly process switching behavior in the System V Message Queue is avoided. The messages are also stored in the order of arrival. As a result, if a process asked for a message and there are more than one messages suit the process, only the first arrived message will be retrieved. Figure 5.3 shows the interaction between the sender, the SMQ and the recipient.

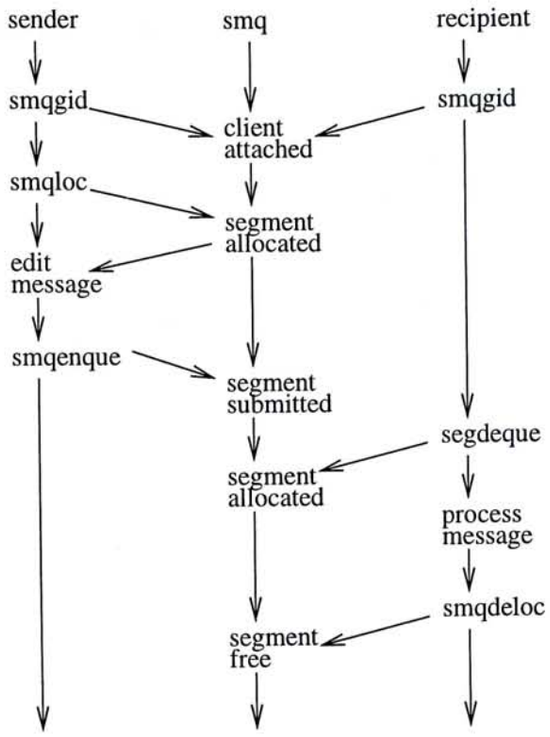


Figure 5.3: Interaction between sender, SMQ and recipient

5.2 SCSI Hardware & Device Driver

Each host in the IRI System is required with a *host adapter* and connected to a *SCSI Bus*. The host adapter controls the connection establish and and data transfer. It is also responsible for closing a connection after data transfer completed. The Bus, which is a *shared media*, simply connect the hosts for communication. The SCSI host adapter is control is by a device driver. The device driver provides direct access function to the processes that need to use the SCSI hardware. However, in our IRIS system, only the Network Message Routing Layer is allowed to use the SCSI hardware. It will lock out other processes from attempting to access the SCSI device.

The SCSI Hardware Support layer also provides a high speed reliable data link

facility. Message received are guaranteed to be error free by the SCSI hardware. The order of message presented to the upper layer is the same as the order of message received from the Bus port (a FIFO queue).

5.2.1 SCSI Bus Operations

On a SCSI Bus, the devices usually have to play one of the two roles: *initiator* and *target*. An initiator determines the command to be executed on the bus. It could be data transfer, device control and inquiry and etc. The target device perform and guard the progress of the command.

In the ANSI SCSI-II specification [GeS93], an number of bus width and transfer rate have been specified. A number of different devices have been specified. In our project, we use a SCSI Bus of which the bus width is 16bit and the transfer rate is 10MHz. Hence, the maximum possible data rate is 20MByte/sec.

The specification also defined a number of the command that could act on the devices. The device type of the hosts is *Processor Devices*, which is a very general device class. The only command associated by the processor device are send and receive data across the bus. In our implementation, only send command is supported. It also provides two entry points, send and receive, for the client processes to access it directly. Only the multiplexer will use the entry points.

The complete protocol involves 10 different *phases*. At any given time, the SCSI bus can only be in one phases. They are:

BUS FREE When the SCSI bus is not being used by a device, it remains in the BUS FREE phase.

ARBITRATION Which device obtains the control of the bus after a BUS FREE is determined in the ARBITRATION phases. If a device wishes to arbitrate for the bus, it activates its SCSI ID on the bus. At the end of the arbitration phase, the device looks at the bus to see if a SCSI ID greater than its own has been activated. The device with the higher ID wins the arbitration.

SELECTION After an initiator wins the arbitration phase, the SELECTION phase takes place. During the selection phase, a connection is established with the desired target by activating the targets SCSI ID on the bus. All devices have to check whether their SCSI ID is asserted on the bus.

RESELECTION The RESELECTION phases allows a target to reconnect to the initiator to complete a command. The SCSI 2 specification allows a device to disconnect the initiator when it is busy. When it is free, it have to reconnect the initiator to finish the command. As a result, the bus could be free for other transmissions.

MESSAGE IN/OUT In the MESSAGE phase the initiator and the target negotiate the transfer options and inform each other about errors occurred during the session.

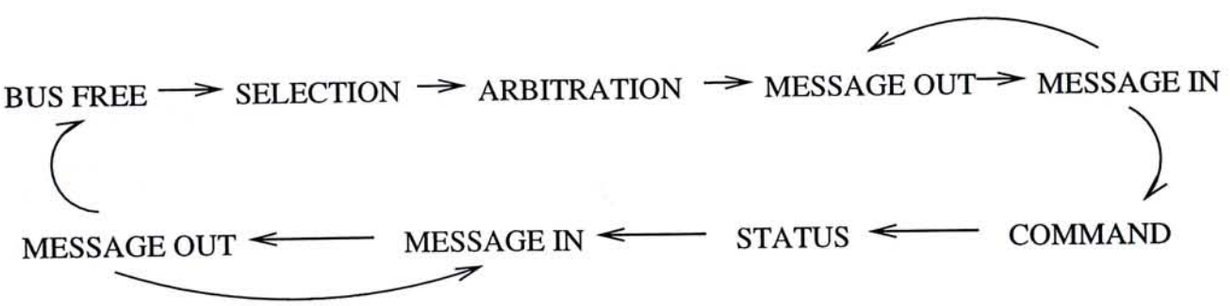
COMMAND The COMMAND phase is used by the initiator to send the SCSI

command to the target. After receiving all the command bytes, the target carry out the command as described in the specification.

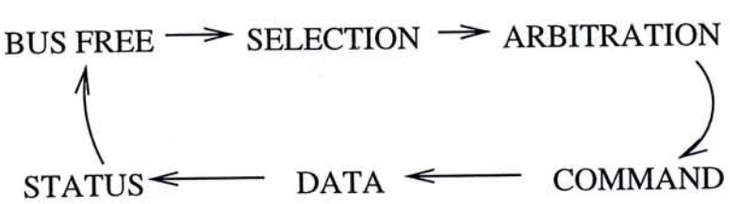
DATA IN/OUT In the DATA phase, control information and user data are exchanged between target and initiator.

STATUS When a command is completed, the target uses the STATUS phase to send status information to the initiator. The information describes the ending status of the command.

There are generally 9 phases change carried out on the bus in a general data sending operation, as depicted in fig 5.4a. In our implementation, only 6 phases changes are carried out (fig 5.4b.) As a result, much of the overhead is saved.



a. General SCSI phase change



b. SCSI phase change in IRI System

Figure 5.4: Simplified SCSI phase changes in sending data

Each session of data transfer on the SCSI Bus is guarded by several phases. The most general phases which exist in most of transactions. These phases includes

BUS FREE, ARBITRATION, SELECTION, RESELECTION, MESSAGE, COMMAND, DATA, STATUS MESSAGE. However, in our system, only several of them are implemented. It is to cut the un-necessary bus phases so as to reduce the time spend on each connection.

Direct access is supported through the entry point provided in the device driver. Blocked mode receive and non-blocked mode send operation is supported. Within the SCSI Hardware Support module, a maximum of 16 unfinished command (sending) could be queued and a maximum of 16 received message could be kept. However, a message received from the remote host would be send to the Local Message Routing Layer immediately after it is received.

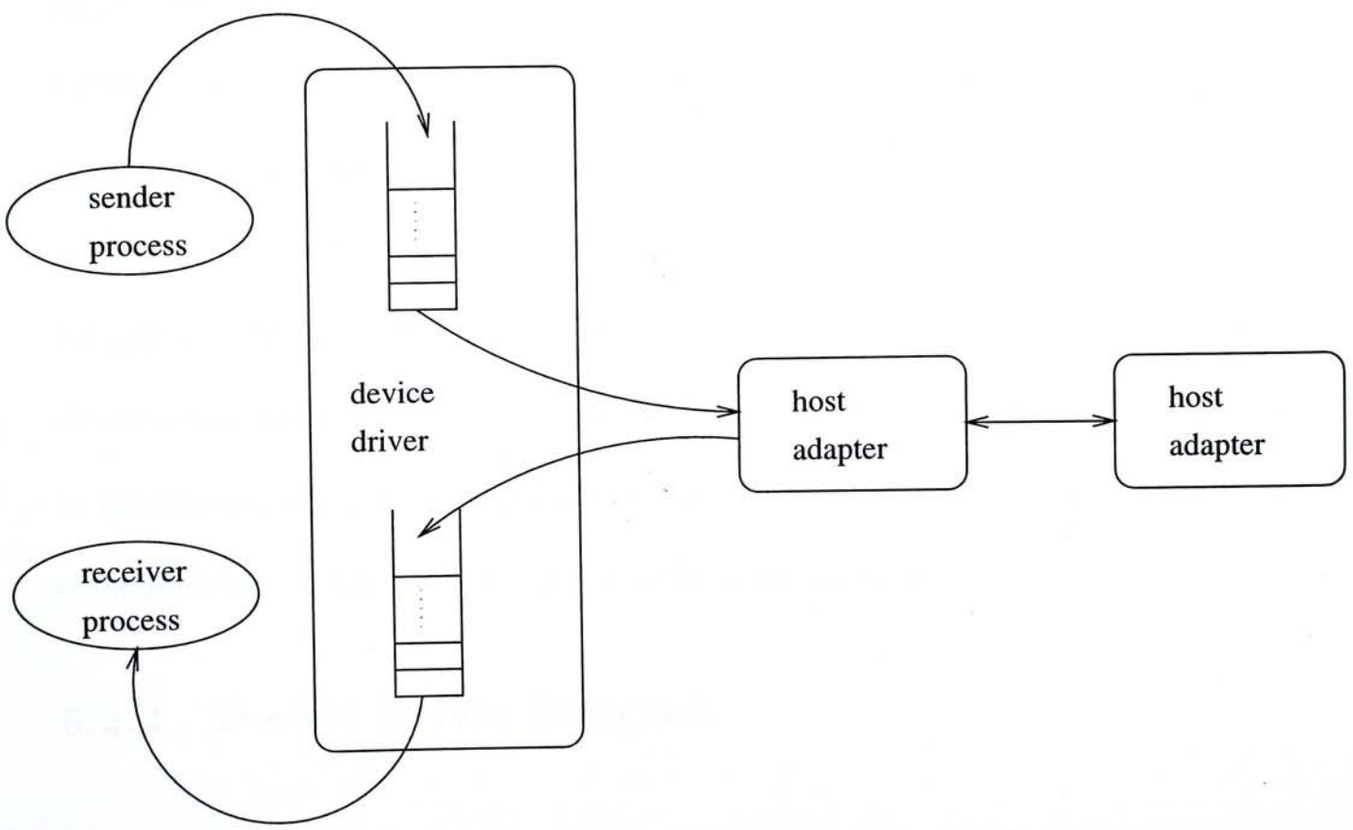


Figure 5.5: Interaction between sender/receiver, device driver and host adapter

In each session of the data transfer, the BUS FREE phase comes first. This indicates that there is nobody using the bus. The host adapter, after receiving a request for sending data, will wait until the BUS FREE. Then it ARBITRATE the bus by inserting its own SCSI ID on the data bus. In the SELECTION phase, the host adapter with the highest order SCSI ID will win the bus. The adapter SELECT the target device for a data transfer. The Initiator then sends the COMMAND to the target. Within the command, the length of data in bytes is encoded. The target than starts the DATA OUT phase and receives the byte(s) from the initiator. What follows is the STATUS phase in which the target reports to the initiator about the result of the transfer. If STATUS is okay the target will put an end to the session by FREEING the BUS. However, if the target returns an error STATUS, the transfer will be regarded as failure and restart (figure 5.6). In the process, all the MESSAGE phases are removed. The purpose of the MESSAGE phases before the COMMAND phases allow the two parties to negotiate the transfer options. In our system, all the transfer options are preset. These options are described in table 5.1. The MESSAGE Phases after DATA Phases allows target to inform initiator for any unpredicted errors. In our system, any unpredicted are concluded by a FAIL STATUS followed by a transmission.

5.2.2 Device Driver Internals

The complete SCSI device driver consists of two parts: *High Level Device Driver (HLDD)* and *Low Level Device Driver (LLDD)*. The HLDD manages a message

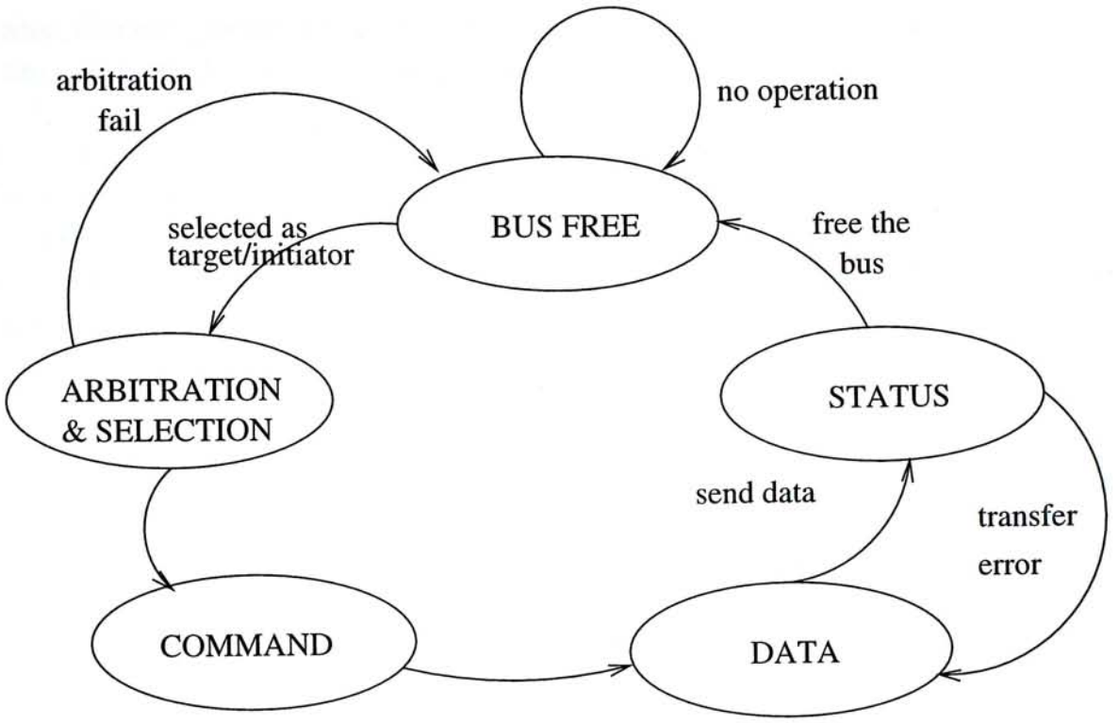


Figure 5.6: State diagram of host adapter

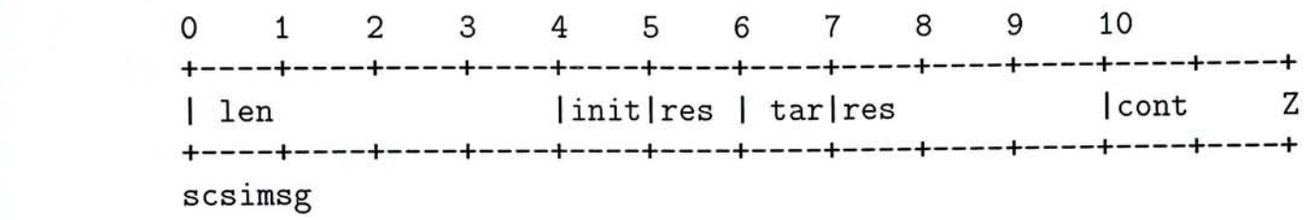
queue and a command queue. The LLDD controls the bus according to the SCSI commands from the high level device driver. It also response to connection from the remote SCSI Adapters.

The direct entry points are provided by the High Level Device Driver. They are:

Bus Phases	IRI Phases	Time
BUS FREE	Yes	1.6 us
ARBITRATION	Yes	2.4 us
SELECTION	Yes	200 ms max
MESSAGE OUT	No	progr.
MESSAGE IN	No	progr.
COMMAND	Yes	progr.
DATA	Yes	progr.
STATUS	Yes	30 ns
MESSAGE IN	No	progr.
MESSAGE OUT	No	progr.

Table 5.1: SCSI BUS Phases in IRI and Normal Process

```
int ahc_direct_send (scsimsg * msg);
int ahc_direct_recv (scsimsg * msg);
```



The scsimsg structure is used as below:

len - length of message content excluding the 10 bytes information header

init - SCSI ID of the initiator, i.e. the SCSI ID of the adaptor on the local host.

tar - SCSI ID of the target, i.e. the SCSI ID of the adaptor on the remote host.

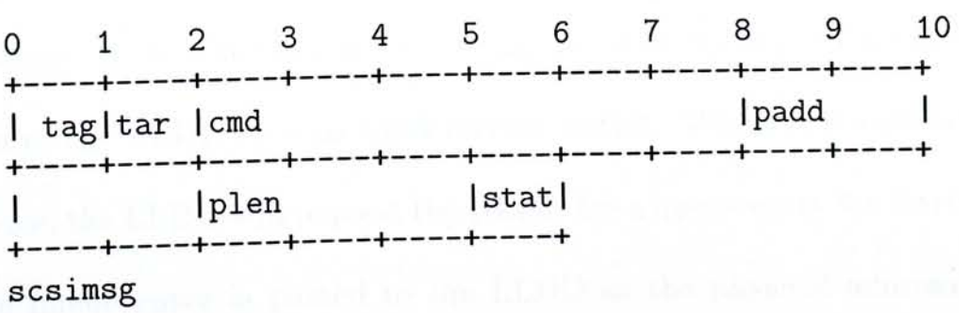
res - reserved bytes. These bytes will not be referenced or modified by the SCSI drivers.

To send a message, the sender uses the `ahc_direct_send` to invoke the HDLL. These High Level Device Driver (HLDD) will than compose a *SCSI Control Block (SCB)* basing on the control information from the sender. The size of message content in the current implementation is limited to 4086 bytes. A queue of 16 entries is used to store the outgoing messages. The sender will return from the `ahc_direct_send` operation immediately after the device driver has make a copy of the required information. However, if the command queue is full, the sender will be blocked until an entry is available. In the current implementation, the control

information together with the message content will be copied into a buffer. This design frees the sender from having to be blocked until the message is received by the remote host. If the local memory space of the process is used instead for copying the message into a buffer, the sender have to be blocked to avoid the memory space being modified. The demerit of this design is that addition memory copies is necessary.

The received message queue has a total of 32 entries. Each time a message comes, the LLDD will request an entry from the HLDD. The HLDD stores the complete message into the queue. The size of each queue entry is 4096 bytes. In order words, the maximum size of the message is 4086 bytes plus the 10 bytes control information. The incoming buffer hold all the incoming message because it is impractical to have the initiator and the target lock the bus waiting for process to receive a message. However, this cause another memory copy operation.

The LLDD is the actual part of the device driver that controls the behavior of the SCSI Link. For sending a message, it uses the information in the SCB to arbitrate a connection. The SCB is of the following structure:



tag - tag is used store the location of the SCB in the queue.

tar - the SCSI of the target device to be in the connection

padd - the physical address of the memory location that contains the message

plen - the physical length of the message

stat - return status of the command after the transfer is finished

tag - tag is used store the location of the SCB in the queue.

To send a message, the HLDD sends the SCB to the LLDD. The LLDD use the *tar* field to beat for a connection. The LLDD will keep beating for the role of initiator as long as there is SCB in the SCB list. When it wins a session, it will pass the peer LLDD the *cmd*. When all the negotiate is finished, the *padd* and the *plen* is used to drive the DMA in the DATA Phase. When the transfer is completed, the target will send it the transmission status (OK or ERROR). Finally, the LLDD informs the HLDD of the complete of the transmission. The HLDD uses the value in the tag to remove the SCB in the SCB list.

To receive a message, the LLDD is usually in a state that ready to be connected as a target. However, when the incoming queue of the HLDD is full, the HLDD will inform the LLDD to wait until further notice. When it is connected as a target device, the LLDD will request the HLDD for a queue entry for storing the message. The queue entry is passed to the LLDD as the physical address. The physical address will be used to drive the DMA in the DATA Phase. The LLDD will return

a transfer OK status to the initiator if not parity error is detected. Otherwise, an ERROR status will be sent and the message will be retransmitted. After the transmission is finished, the LLDD will inform the HLDD that the message is ready to be retrieved by the client processes. Fig 5.7 shows the interaction between the sender, SCSI device driver and the recipient.

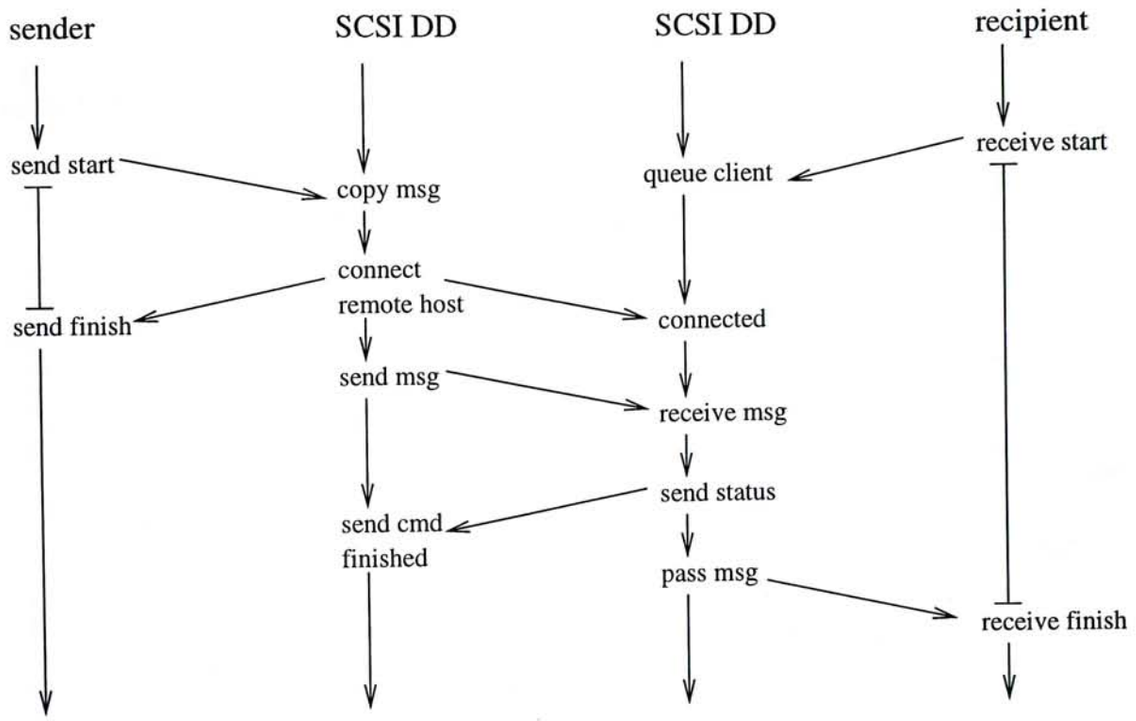


Figure 5.7: Interaction between sender, SCSI device driver and recipient

In PC systems, each of the hardware device has a address, which is usually mapped to a physical address in the memory space. The HLDD communicates with the LLDD using the SCSI adapter's *physical address*. The LLDD runs inside the SCSI adapter within which there are a number of registers and scratch memory. The scratch memory holds a number of variables that is used by the LLDD to guard its operation. The variables can also be visited by the HLDD for monitoring the operation and communicating with the LLDD. The HLDD also uses the variables

to inform the LLDD of its status. The most important variable is the MQSTATUS. The HLDD uses it to select the incoming message queue's status: FULL or READY. If the MQSTATUS is FULL, the LLDD will disable its capability of being selected as a target device.

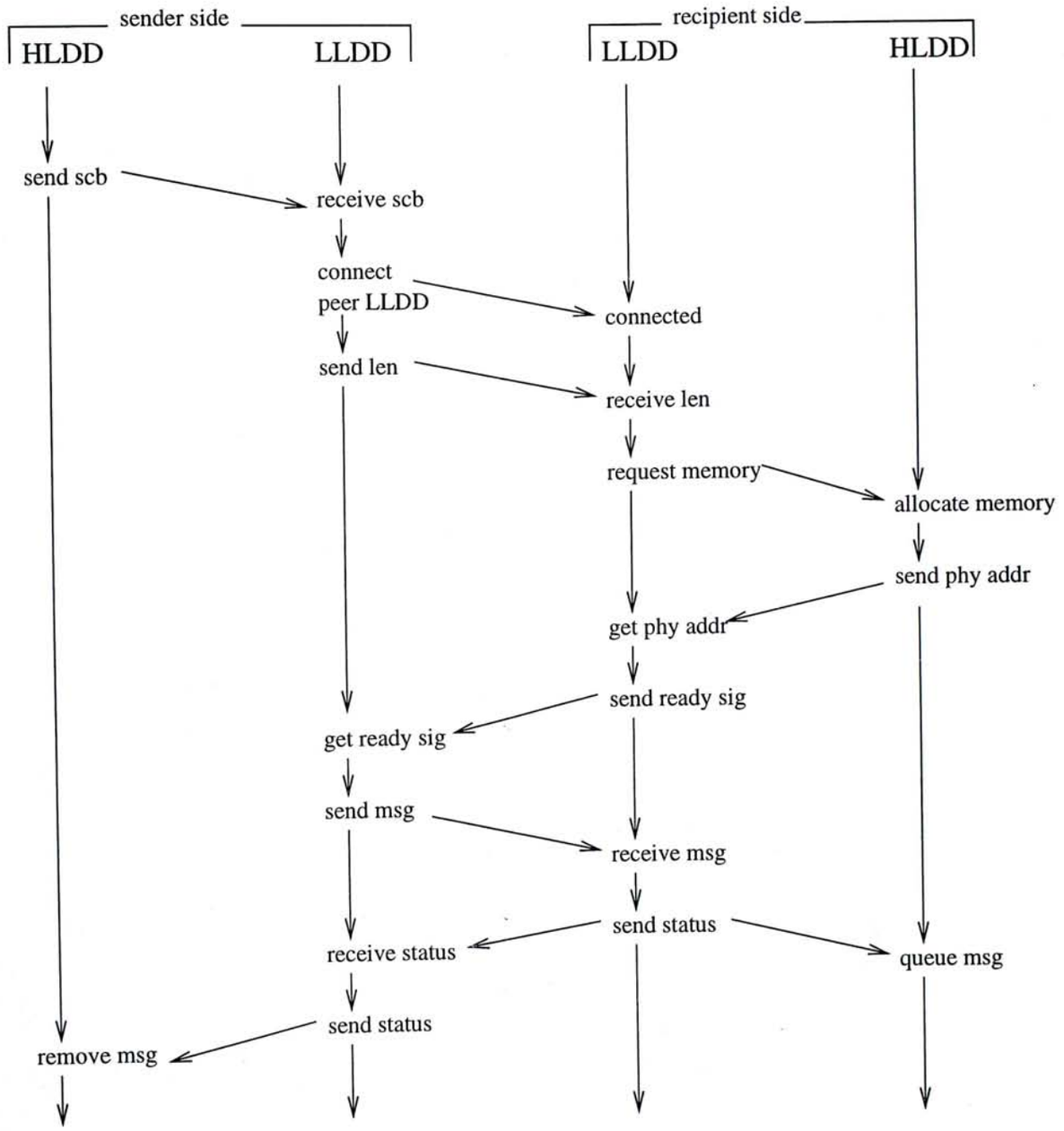


Figure 5.8: Interaction between SCSI HLDD and LLDD

To communicate with the HLDD, the LLDD uses the PC *interrupt system*. When an event happens, the LLDD puts a code in its variable, INTCODE. Then it

invokes an interrupt which will be caught by the HLDD. The HLDD reads the INTCODE and other related variables and react to those events. Fig 5.8. depicts the detail interaction between the HLDD and the LLDD.

Chapter 6

Performance

In performance comparison, PVM is used as a control. Both the PVM and the IRI systems are run on the same machines, running the same OS. The only difference is that the PVM uses the Ethernet for network message passing whereas the IRI uses the SCSI Link.

The first benchmark program measures the time of message passing between two processes. The *round trip time* is measured. In each measurement, a message is sent from a master process to a slave process and returned from the slave to the master. The time difference between the send and receive with respect to the master process is measured.

Table 6.1 shows the performance of IRI and PVM in transferring local message. For each size, the average of 20 transmission was taken and compared. The IRI system outperforms the PVM system in local message passing by 8 to 10 times. Figure 6.1 plots the performance of the two systems.

size	iri	pvm	pvm/iri
200	232.2	1874.2	8.069
400	236.6	1924.0	8.131
600	238.8	1954.6	8.185
800	237.7	1991.7	8.379
1000	236.5	2029.8	8.582
1200	238.1	2082.2	8.745
1400	236.8	2102.7	8.879
1600	237.9	2175.0	9.142
1800	239.6	2181.5	9.102
2000	237.9	2191.2	9.208
2200	239.1	2329.7	9.741
2400	239.5	2283.9	9.534
2600	240.6	2427.4	10.088
2800	238.0	2373.2	9.971
3000	238.0	2451.1	10.296
3200	239.3	2408.5	10.062
3400	239.7	2461.6	10.267
3600	238.6	2484.1	10.408
3800	242.1	2455.2	10.139
4000	240.1	2509.4	10.449

Table 6.1: IRI and PVM Local Message Passing

In the IRI system, the time required for sending of local message is nearly constant. It is because the SMQ system was employed which takes the advantages of shared memory. The PVM is about 8-10 times slower that than the IRI system. It is accounted by the use of TCP/IP in the transmission of PVM messages.

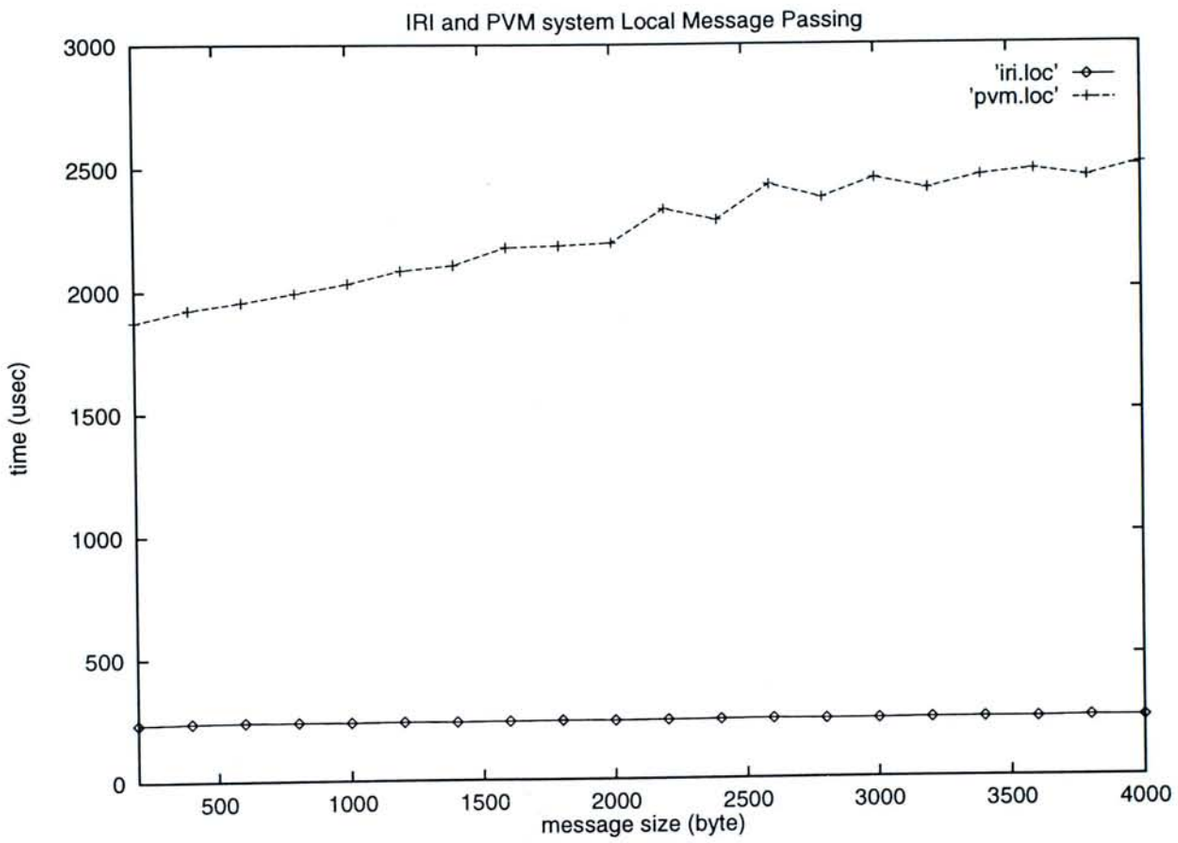


Figure 6.1: IRI and PVM system Local Message Passing

The second benchmark result shows the difference in performance between IRI and PVM in transmitting network message. The IRI system outperforms the PVM system by 5-8 times (table 6.2). In measuring this set of data, the latest version of the IRI system with SCSI Link supported is used. It accounts for the better performance delivered. However, due to the quality of the device driver, the raw speed of the SCSI Link is not yield. If the raw speed is achieved, the IRI system should be 16 times faster than the PVM system (20 Mbyte against 10 Mbit.)

size	iri	pvm	pvm/iri
200	777.4	3953.5	5.085
400	802.0	4619.9	5.759
600	825.9	5300.6	6.417
800	862.7	4961.5	5.750
1000	1017.0	5658.2	5.563
1200	1047.4	5992.5	5.720
1400	1074.2	6653.1	6.193
1600	1096.7	6926.1	6.314
1800	1144.7	7484.3	6.538
2000	1148.7	7782.9	6.775
2200	1174.4	8159.9	6.948
2400	1202.6	8240.6	6.852
2600	1246.0	8488.5	6.812
2800	1285.7	8967.2	6.974
3000	1477.5	9763.3	6.607
3200	1598.0	9887.9	6.187
3400	1612.5	10466.5	6.490
3600	1638.8	10866.5	6.630
3800	1659.8	11362.6	6.845
4000	1738.0	11835.3	6.809

Table 6.2: IRI and PVM Network Message Passing

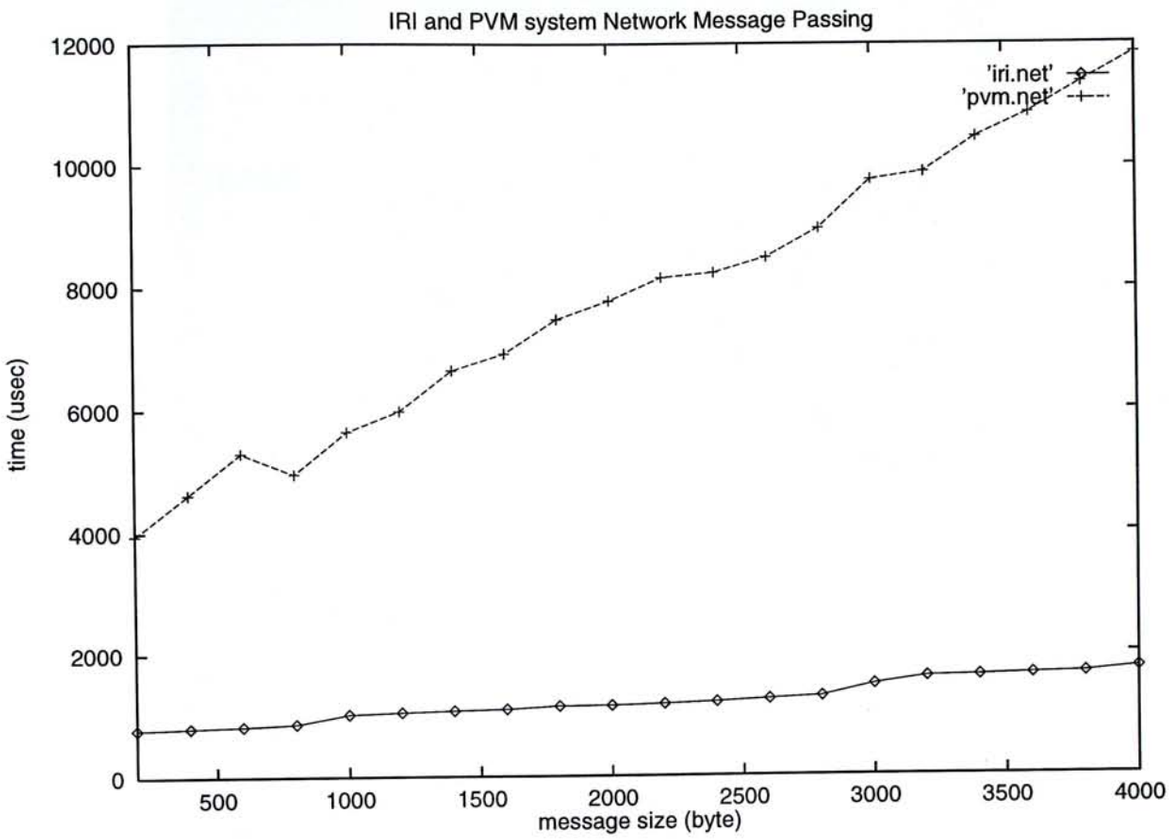


Figure 6.2: IRI and PVM system Network Message Passing

The second benchmark program is a fractal image painter (fig 6.3.) In the program, we have employed a master slave model. A dynamic job allocation method is adopted. A slave process after finish a job will be assigned a new one immediately if there exists non-finished jobs. The program was chosen because it allows a dynamic change of the granularity of jobs. The task of drawing an image is divided into a number of tasks of drawing line segments. Each segment is drawn as instructed by three parameters. The job nature can also be varied using the parameters. The parameters are: line size, iteration, and region. Out of the three parameters, the line size and the iteration are of high significance in affecting the job nature.

A line is computed using the following routine:

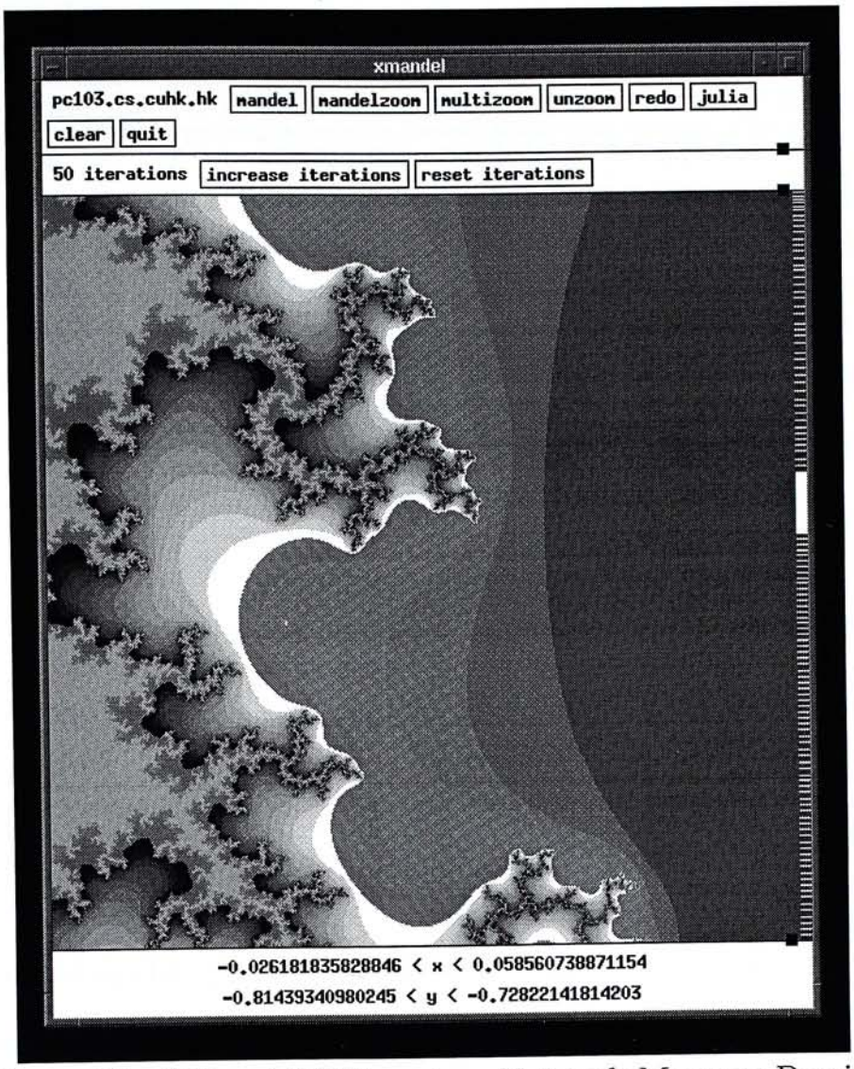


Figure 6.3: IRI and PVM system Network Message Passing

```
        wwidth = edx - stx; x +=stx*incrx;
        for (ix = 0; ix < (long)wwidth; ix++, x+= incrx) {
            wx = x; wy = y;
            for (i = 0; i < iter; i++) {
                    if ((wx2 = wx * wx) + (wy2 = wy * wy) > 4) break;
                    wy = 2 * wx * wy + y;
                    wx = wx2 - wy2 + x;
            }
            dp[ix] = i + bias;
        }
where
wwidth - width of this line segment
edx    - end x coordinate
sdx    - start x coordinate
ix     - current pixel
iter   - number of iteration
dp     - the line of pixel
```

Each job is consists of two messages. The first message is sent form the master to

line size	PVM system	IRI system	PVM/IRI
5	43810331	7779623	5.631420828
10	43877489	7786936	5.634756597
20	44587053	8272623	5.389711703
50	44470087	8710225	5.105503819
100	44782622	9087292	4.928049192
150	45398492	9161860	4.95516107
200	45305556	10068496	4.49973422
300	46913224	11110879	4.222278363
400	47695336	12056309	3.956047908
500	48265225	12895091	3.742914649
600	49133385	13747132	3.57408258
700	49805835	14390316	3.461066109
800	49714329	15212452	3.268002358
900	50384801	15783069	3.192332302
1000	50856950	16637456	3.056774425

Table 6.3: Effect of Line Size on IRI and PVM system

the slave, specifying the information about the line segment to be drawn. It is a fixed size message. The second message is returned from the slave to the master. The message consists of the coordination of a line and pixels of the line. The *Line Size* affect the message size and the CPU required. *Iteration* determines at most how many iterations is allowed to calculate a pixel value. It is directly affecting the CPU resource required. Given a fixed region to be drawn:

$$\text{Message Size} \propto \text{Line Size}$$

$$\text{CPU Resource} \propto \text{No of Iteration} \times \text{Message Size}$$

The third benchmark result shows the message passing performance of the IRI system and the PVM system where computation required is extremely low. The iteration is set to 1. The Line Size varies as shown in the table .

The fourth benchmark result shows the performance of mixing network and local

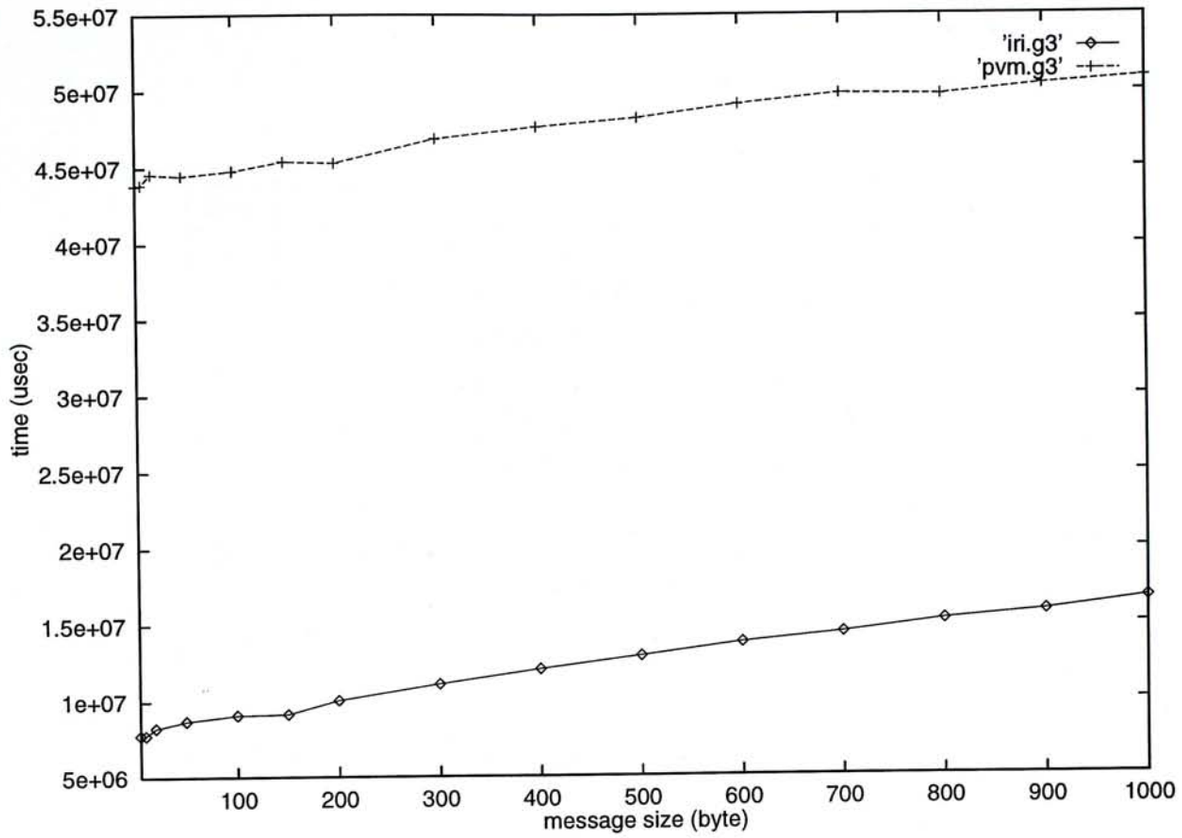


Figure 6.4: Effect of Line Size on IRI and PVM system

messages. Different iteration and line size combinations are tested.

Fig 6.5 shows the difference in performance between IRI system and PVM system, under different computation size and message size requirement. When both the computation and message size are small, the task is a message intense one. These kind of jobs are those that generate a large volume of small messages. When both the message size and the computation size are large, the task is a CPU bound one. These kind of jobs does not require high bandwidth communication channel for information exchange. In fact, this is the kind of jobs that gain benefits from the previous generation of cluster environment parallelization tools.

iter	linesize			
	150	250	520	800
5	14537094	17395090	24265680	30416607
10	17539136	21691296	31799799	42059184
15	19929121	24830446	37656002	49699217
20	21912337	27864458	43127790	57090424
25	23416689	30447998	47539143	63675922
35	26703701	34732734	56361411	74290676
50	30945499	40896462	60008629	83018185
75	37283188	50953839	77161725	93716243
100	43453257	60561515	83225172	104907525

Table 6.4: Effect of Line Size and Iteration on IRI system

iter	linesize			
0	150	250	520	800
5	47922278	49452397	53600694	57046218
10	48921923	50963710	57034538	62434279
15	49733518	52163178	59908407	66259063
20	50430379	53417335	62708720	69603193
25	51159201	54665610	64898710	72099786
35	52516388	56476293	68188154	77074854
50	54537658	59875690	72995348	84125688
75	57761665	64131649	80739559	95348021
100	60681627	68080109	88286150	106273552

Table 6.5: Effect of Line Size and Iteration on PVM system

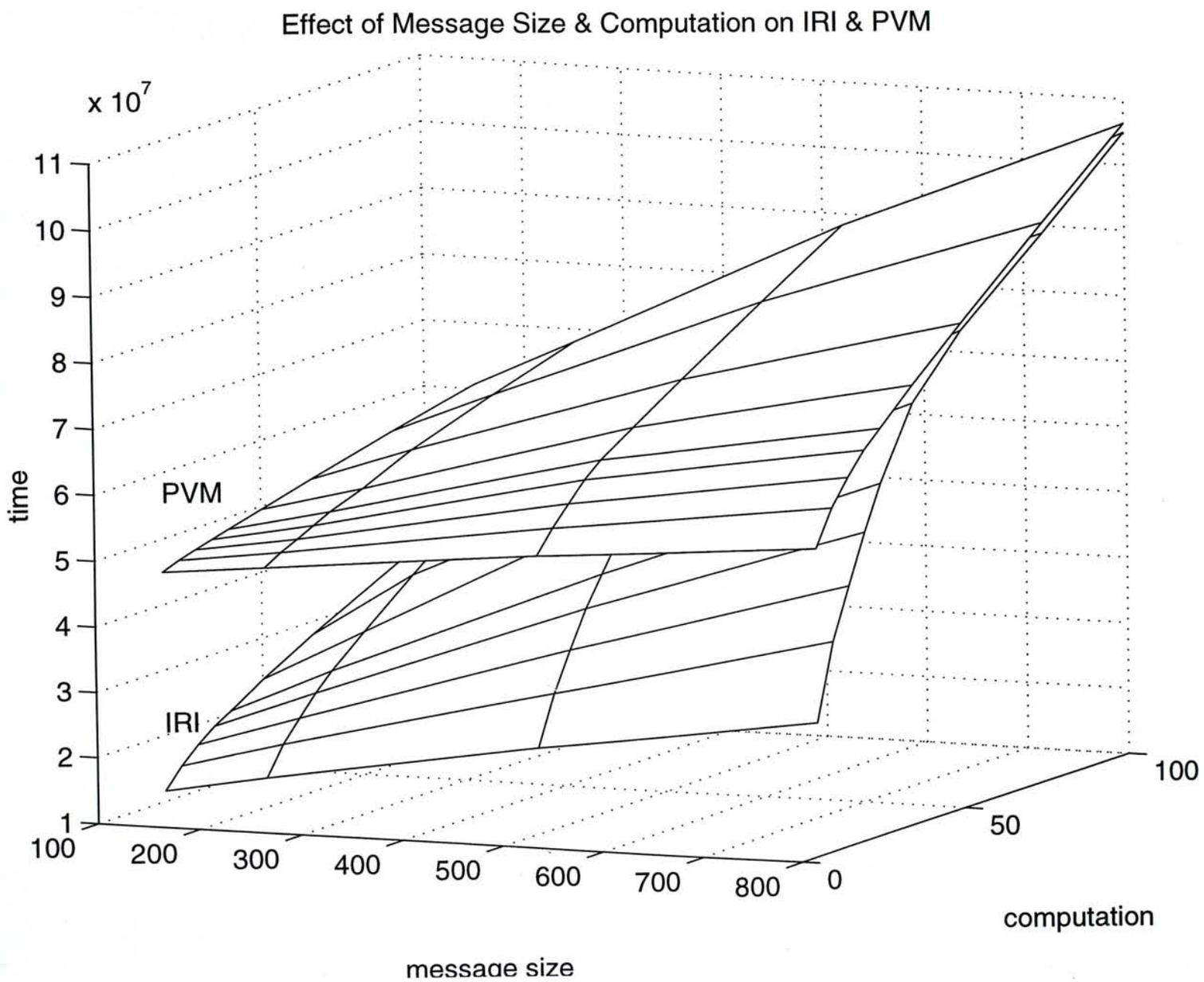


Figure 6.5: Effect of Message and Computation Size on IRI and PVM

Chapter 7

Conclusion

7.1 Summary of Our Research

In this thesis, a multiple computer infrastructure has been has been proposed. Three sets of components are designed and implemented one by one based on the need at different level of the infrastructure:

CommUnication Protocol (CUP): It is observed that parallel and distributed applications have a common needs in communication methods and system services. The services include the creation and deletion of processes and various kinds of resource request. Examples of communication methods are sending, receiving, multicasting and group operations. The CUP specifies a small and simple set of interfaces for communication and services. Complex operations could in turn be built using the set of specified interfaces.

Multiple Computer Architecture (IRI): A three layered multiple computer architecture has been proposed and built. Each layer provides different aspect of service. A upper layer is served by a lower layer. The internal of a layer is hidden from the other layers. The *Application Support Layer (ASL)* provides

facilities for the clients to request for system services and communication. The *Local Message Routing Layer (LMRL)* passes messages for clients on the same host. The *Network Message Routing Layer (NMRL)* passes messages for the LMRL on different hosts.

System Supporting Facilities: Two system Facilities supporting the operation the multiple computer architecture is developed. These facilities are not regarded as part of the multiple computer infrastructure. Other programs may also these facilities.

As an conclusion, we have built a low cost multiple computer system, which is about 6 times faster than the traditional multiple computer system, for tighter-coupled applications. The system is built using off-the-shelf hardwares and free softwares. It is ready to be used by some parallel applications and algorithms. The communication protocol provides a easy mean for porting existing applications to our system. The system also provides a platform for further studies of multiple computer architecture.

7.2 Future Researches

In this thesis, a high performance multiple computer architecture is purposed and implemented.

Distributed Shared Memory (DSM) DSM paradigm is an active research topic in the area of parallel and distributed processing. DSM systems usually provide and simple and easy to program environment when compared with message passing programming environment. The architecture could be used

to studies the behavior of DSM on tighter-coupled systems.

Message Routing Algorithm To improve the performance of the system, different mechanism on routing messages could be applied. In the current implementation, the messages are directly sent to the destinations. However, mechanisms could be applied to filter local messages and network messages especially when multicasting is in concerned.

High Speed Network In the current implementation, a 20 MByte/sec SCSI link is used as the communication media. As technologies advances, faster SCSI link is available. 40 MByte/sec SCSI is ready, and 80 MByte/sec should be right on the market soon.

Network Architecture It is expensive to use the top speed BUS for communication. Besides, some parallel algorithms does not demand equal bandwidth for all the processing elements, if the partition of job is well designed. Different network architecture could be tested for these algorithms.

Bibliography

[Aic95] *AIC-7870 PCI bus master single-chip SCSI host adapter data book,* Adaptec, Inc, 1995.

[GeB95] Geist, A., Beguelin, A., Dongarra, W., Manchek, R., Sunderam, V., *PVM 3 User's guide and Reference Manual,* Oak Ride National Labratory, 1995.

[BrD94] Bruns, G.D., Daoud, R.B., Vaigl, J.R. *LAM: An Open Cluster Environment for MPI,* Supercomputing Symposium '94, 1994.

[CaG89] Carriero, N., Gelernter, D. *Linda in contex,* Communication of the ACM, vol 32, pp 444-458, April 1989.

[Cat95] Catanzaro, B. *Multiprocessor System Architectures,* Sun Microsystem, 1995.

[ChG94] Chao H.J., Ghosal, D., Saha, D., Tripathi, S.K. *IP on ATM local area networks,* IEEE Communications Magazine, pp 52-59, August 1994.

[DaG88] Darema, Frederica., George, D.A., Norton, V.A., Pfister, G.F. *A single-program-multiple-data computational model for epex fortran,* Paralle Computing, vol 7, pp 11-24, 1988.

[DeC89] DeCegama, A.L. The technolog of parallel processing - parallel processing architectures and VLSI hardware Volume 1, Prentice Hall, 1989.

[GeS93] Geist, G.A. Sunderam, V.S. *The evolution of the PVM concurrent computing system.* Proceedings of the 26th IEEE Compcon Symposium, pp 471-478, Feb 1993.

[Gol93] Golden, R.G., *Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory,* Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems, pp 58-67, Oct 1993.

[HwM96] Hwang, C. McKinley, P.K. *Communication issues in parallel computing across ATM networks,* IEEE Parallel and Distributed Technology, to appear.

[KhN93] Khalidi, Y.A., Nelson, M.N. *The Spring virtual memory system,* Sun Microsofts Laboratories, Report SMLI TR-93-9, Feb 1993.

[LeM90] Leffler S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S. *The design and implementation of the 4.3BSD UNIX operating system,* Addison-wesley, 1990.

[Mar94] Martin, R.P. *HPAM: An Active Message Layer for a Network of HP Workstations,* Hot Interconnects, 1994.

[MPI94] *MPI: A Message-Passing Interface Standard (Version1),* May 1994. Message-Passing Interface Forum.

[OkD90] O'Keefe M.T., Dietz H.G. *Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors,* 1994 Internation Conference on Parallel Processing, pp 93-96, 1990.

[Paj91] Pajari, G. *Writing Unix device drivers*, Addison-wesley, 1991.

[Rag93] Rago, S.A. *UNIX System V Network Programming* Addison-Wesley, 1993.

[ANS94] *SCSI-2 Specification,* Document X3.131-1994,ANSI.

[Sch95] Schmidt, F. *The SCSI bus and IDE interface - protocols, applications and programming,* Addison-Wesley, 1995.

[Ste94] Stevens W.R. *TCP/IP Illistrated Volume 1 - The Protocols* Addison-wesley, 1994.

[Sur93] Surderam V.S. *Empirical Analysis of Overheads in Cluster Environments*, Emory University, 1993.

[WhA94] White, S., Alund, A., Sunderam, V.S. *Performance of the NAS Parallel Benchmarks on PVM Based Networks,* Emory University, Report RNR-94-008, May 1994.

[WrS95] Wright, G.R., Stevens W.R. *TCP/IP Illistrated Volume 2 - The Implementation* Addison-wesley, 1995.